CS315 - Homework 1 Report

Halil Arda Özongun

ID: 22202709

Department of Computer Engineering

Bilkent University

Ankara, Turkey

April 7, 2025

1 Operations in each language

1.1 Dart

Dart is an object-oriented programming language with statically typed system and JIT (Just-In-Time) compiler. In Dart there is no such thing as an array; instead there is a class `List` which can be dynamic or fixed length. `Lists` are linear data structures that can expand, hold different types of objects, and can be manipulated.

- **What types are legal for subscripts?**
  In Dart, array subscripts must be integer. Only integer types can be used after an explicit conversion if they're not int by default. Other types are illegal to use, and gives compile time error.

```
print("1: Legal types for subscripts");
List<int> arr1 = [1, 2, 3, 4, 5];
print(arr1[3]);
// print(arr1[3.0]);     Compile time error
// print(arr1["1"]);     Compile time error
// print(arr1[true]);    Compile time error
/*double floatIndex = 2.8;
print(arr1[floatIndex]); // Compile time error */
```

  Printed Output:

```
1: Legal types for subscripts
4
```

- **Are subscripting expressions in element references range checked?**
  Yes, there is range checking. It is checked at runtime and if the number is out of range an error is thrown.

```
print("\n\n2: Range checking in subscripting expressions:");
// print(arr1[10]); // Runtime error
// print(arr1[-1]); // Runtime error
int index = 4324;
// print(arr1[index]);  // Runtime error
```

  Printed Output:

```
2: Range checking in subscripting expressions:
```

- **Are ragged multidimensional arrays allowed?**
  Yes, Dart allows any kind of multidimensional arrays.

```
print("\n\n3: Ragged multidimensional arrays:");
List<List<int>> ragged = [
[1, 2, 3],
[4, 5],
[6, 7, 8, 9]
];

print(ragged);
print(ragged is List<List<int>>);
print(ragged[0] is List<int>);
```

Printed Output:

```
3: Ragged multidimensional arrays:
[[1, 2, 3], [4, 5], [6, 7, 8, 9]]
true
true
```

- **Can array objects be initialized?**
  Yes. When creating an array in Dart, you can give the elements as a list. This list can be heterogeneous. Another way to initialize arrays is to use the `List.filled` method to give the size of the arrays and initialize the entire array with the same value. It can also be filled with a function for the specified size with the `List.generate` method. Dart also supports unmodifiable lists via `List.unmodifiable`, and lists can be built dynamically using the spread operator `...` and comprehensions with loops or conditions.

```
print("\n\n4: Initialization of arrays:");
var init1 = [1, "ataturk", 3, true];
print("init1: $init1");
var init2 = List.filled(5, null, growable: false);
print("init2: $init2");
var init3 = List<int>.generate(8, (i) => i *  i);
print("init3: $init3");
var init4 = List.of([5]);
print("init4: $init4");
const init5 = [1, 2, 3, 4, 5];
print("init5: $init5");
var init6 = List.unmodifiable([1, 2, 3, 4, 5]);
print("init6: $init6");
var init7 = [...init1, ...init2, ...init3];
print("init7: $init7");
var init8 = [for (var i = 0; i < 5; i++) i * i];
print("init8: $init8");
var init9 = List.empty(growable: false);
print("init9: $init9");
```

Printed Output:

```
4: Initialization of arrays:
init1: [1, ataturk, 3, true]
init2: [null, null, null, null, null]
init3: [0, 1, 4, 9, 16, 25, 36, 49]
init4: [5]
init5: [1, 2, 3, 4, 5]
init6: [1, 2, 3, 4, 5]
init7: [1, ataturk, 3, true, null, null, null, null, null, 0, 1, 4, 9, 16, 25, 36, 49]
init8: [0, 1, 4, 9, 16]
init9: []
```

- **Are any kind of slices supported?**
  Yes. In Dart, a slice can be created by specifying the start index for slicing. You can also specify the end index if desired. It is also possible to take the first N elements, or take after the first N elements. It is also possible to create the desired slice by filtering the array.

```dart
print("\n\n5: Array Slice:");
arr1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
var slice1 = arr1.sublist(2, 5); // (start,end)
print("slice1 : $slice1");
var slice2 = arr1.sublist(3); // start
print("slice2 : $slice2");
// First N elements
var slice3 = arr1.take(4).toList();
print("slice3: $slice3");
// skip first n elements
var slice4 = arr1.skip(5).toList();
print("slice4: $slice4");
// Filtering
var slice5 = arr1.where((x) => x > 5).toList();
print("slice5: $slice5");
```

  Printed Output:

```
5: Array Slice:
slice1 : [3, 4, 5]
slice2 : [4, 5, 6, 7, 8, 9, 10]
slice3: [1, 2, 3, 4]
slice4: [6, 7, 8, 9, 10]
slice5: [6, 7, 8, 9, 10]
```

- **Which operators are provided?**
  In Dart, we can check some equalities for arrays. One way to do it is using the == operator. Lists are, by default, only equal to themselves [1]. That operator does not compare the elements of the list - it checks if both references point to the same object.

Dart also provides a function called `identical`, which is not an operator, but worth mentioning for comparing with `==`. It checks whether two object references refer to the exact same memory object [1]. Like `==`, it does not compare list contents.

To perform deep equality checks (i.e., comparing contents of lists), Dart provides the `ListEquality` class from the `collection` package. This class offers a method `equals`, which returns `true` if two lists have the same elements in the same order, regardless of whether they are different objects.

Dart supports list concatenation in two ways: using the `+` operator, or using the spread operator `[...a, ...b]`. Both produce a new list by joining the elements of the input lists.

Additionally, Dart supports the `contains` method (similar to an `in` operator) to check whether a list contains a specific value. This returns a boolean result.

```
print("\n\n6: Allowed operations:");

var a = [1, 2, 3];
var b = [4, 5, 6];
var c = [1, 2, 3];

print("a: $a, b: $b, c: $c");

print("a == c: ${a == c}"); // false, different objects
print("identical(a, c): ${identical(a, c)}"); // false, different
    objects

var d = a;
print("a == d: ${a == d}"); // true, same object
print("identical(a, d): ${identical(a, d)}"); // true, same object

Function eq = const ListEquality().equals;
print("ListEquality: ${eq(a, c)}");

// concatenation
var ab = a + b;
print("Concatenation: $ab");

var concat = [...a, ...b];
print("Concatenation: $concat");

// Contains
print("2 in a: ${a.contains(2)}");
print("10 in a: ${a.contains(10)}");
```

Printed Output:

```
6: Allowed operations:
```

```
a: [1, 2, 3], b: [4, 5, 6], c: [1, 2, 3]
a == c: false
identical(a, c): false
a == d: true
identical(a, d): true
ListEquality: true
Concatenation: [1, 2, 3, 4, 5, 6]
Concatenation: [1, 2, 3, 4, 5, 6]
2 in a: true
10 in a: false
```

1.2 Go

Go is a statically typed and compiled programming language. In Go there are two commonly used data structures, slices and maps. Slice is a dynamically sized linear data structure, which provides features like reslicing, appending, etc. Map is an unordered collection of key-value pairs (like hashmap). This slice shouldn't be confused with the one we use in programing languages context, which means "a part of array". Since slice is dynamic sized, we will use array instead of slice. Another important topic to mention is, their decleration or initialization are almost same. The only difference, you have to give the size if you are creating an array (even if you are also initializing).

- **What types are legal for subscripts?**
  In Go, array subscripts must be integer. Only integer types can be used after an explicit conversion if they're not int by default. Other types are illegal to use, and gives compile time error.

```go
fmt.Println("1: Legal types for subscripts: ")
arr1 := [5]int{1,2,3,4,5}
fmt.Println(arr1[3])
var index int64 = 4
fmt.Println(arr1[index])
fmt.Println(arr1[int64(3)])
//fmt.Println(arr1[3.3]) // Compile time Error: invalid array index
    3.3 (type float64)
fmt.Println(arr1[3.0])
// fmt.Println(arr1["3"]) // Compile time Error: invalid array index
    "3" (type string)
var index3 float64 = 3.0
fmt.Println(arr1[int(index3)])
// fmt.Println(arr1[index3]) // Compile time Error: (variable of type
     float64) must be integer
```

  Printed Output:

```
1: Legal types for subscripts:
4
5
4
4
4
```

- **Are subscripting expressions in element references range checked?**
  Yes, there is range checking. For constants it is checked at compile time and for variables it is checked at runtime and if the number is out of range an error is thrown.

```go
fmt.Println("2: Range checking in Subscripting expressions: ")
// fmt.Println(arr1[9]) // Compile time error for constant index out
    of range
```

```
// fmt.Println(arr1[-2]) // Compile time error for negative index
//var index2 = 9
// fmt.Println(arr1[index2]) // Runtime error: index out of range [9]
     with length 5
```

Printed Output:

```
2: Range checking in Subscripting expressions:
```

- **Are ragged multidimensional arrays allowed?**
  No. You can achive ragged multidimensinal structures by using `slices` but you can't use arrays of arrays. If you try to do it, Go fills the places with zero to make it rectangular. If you don't give the size, then it will be slice so again, not a ragged multidimensional array is created.

```
fmt.Println("3: Ragged multidimensional arrays: ")

var multidimensionalArr = [3][2]int{{1,2}, {3,4}, {6,7}}
//var multidimensionalArrOOB = [3][2]int{{1,2}, {3,4}, {6,7,8}} //
    Compile time error.
fmt.Println(multidimensionalArr)

fmt.Println(reflect.TypeOf(multidimensionalArr).Kind())
fmt.Println(reflect.TypeOf(multidimensionalArr[0]).Kind())

var raggedArr = [3][4]int{{1,2,3,4}, {5,6}, {7,8,9}} // Legal. but
    not ragged. It fills with 0
fmt.Println(raggedArr)
fmt.Println(reflect.TypeOf(raggedArr).Kind())
fmt.Println(reflect.TypeOf(raggedArr[1]).Kind())

var raggedSlice = [3][]int{{1,2}, {3,4,5}, {6,7,8,9}} // If size is
    not given, it is array of slices
fmt.Println(raggedSlice)
fmt.Println(reflect.TypeOf(raggedSlice).Kind())
fmt.Println(reflect.TypeOf(raggedSlice[1]).Kind())
```

Printed Output:

```
3: Ragged multidimensional arrays:
[[1 2] [3 4] [6 7]]
array
array
[[1 2 3 4] [5 6 0 0] [7 8 9 0]]
array
array
```

```
[[1 2] [3 4 5] [6 7 8 9]]
array
slice
```

- **Can array objects be initialized?**
  Yes. Go provides many methods for initializing arrays. While initializing, the size must be specified to create an array, if not then slices created. If size is not specified directly like in `arr3` then you should put **...** to assert it is an array. If that argument is not given, a `slice` will be created instead of `array`.

```
fmt.Println("4: Initialization of arrays: ")
var arr2 = [5]int{1,2,3,4,5}
fmt.Println(reflect.TypeOf(arr2).Kind())
var arr3 = [...]int{1,2,3,4,5}
fmt.Println(reflect.TypeOf(arr3).Kind())
var arr4 = [5]int{} // Initialization with only size
fmt.Println(arr4)
fmt.Println(reflect.TypeOf(arr4).Kind())
var arr5 [5]int = [5]int{1,2,3,4,5}
fmt.Println(reflect.TypeOf(arr5).Kind())
```

  Printed Output:

```
4: Initialization of arrays:
array
array
[0 0 0 0 0]
array
array
```

- **Are any kind of slices supported?**
  Yes. Array slicing is supported in Go, but the result is a slice, not an array. You can specify the start and end indices, and Go also supports a three-index slice form: `[start:end:cap]`, where `cap` sets the capacity of the resulting slice. It is possible to not giving any of the start, end or cap arguements.

```
fmt.Println("5: Array Slice: ")
slice1 := arr1[1:3]
slice2 := arr1[1:]
fmt.Println(slice1)
fmt.Println(reflect.TypeOf(slice1).Kind())
fmt.Println(slice2)
fmt.Println(reflect.TypeOf(slice2).Kind())
// slice3 := arr1[1:3:1] //  Compile time error.
slice3 := arr1[1:3:4] // [start:end: max capacity]
fmt.Println(slice3)
fmt.Println(reflect.TypeOf(slice3).Kind())
```

Printed Output:

```
5: Array Slice:
[2 3]
slice
[2 3 4 5]
slice
[2 3]
slice
```

- **Which operators are provided?**

  In Go, the equality operator `==` performs value-based comparison for arrays. Two arrays are considered equal if they have the same length and their elements are equal in the same order. This operator only works on arrays with the exact same type and length. Attempting to compare arrays of different lengths or types will result in a compile-time error.

  To check whether two arrays are located at the same memory address, Go allows the use of pointers with the `&` operator. Two pointers can then be compared using `==` to determine if they point to the same array in memory.

```go
fmt.Println("6: Allowed Operations: ")

a := [3]int{1, 2, 3}
b := [3]int{4, 5, 6}
c := [3]int{1, 2, 3}

fmt.Println("Equality: value-based")
fmt.Println("a == b:", a == b)
fmt.Println("a == c:", a == c)

/*d := [4]int{1, 2, 3, 4}
fmt.Println("a == d:", a == d)*/ // Compile-time error: mismatched
    types

fmt.Println("Adress equality using pointers")
p1 := &a
p2 := &b
fmt.Println("p1 == p2:", p1 == p2)
```

Printed Output:

```
6: Allowed Operations:
Equality: value-based
a == b: false
a == c: true
Adress equality using pointers
```

```
p1 == p2: false
```

1.3 JavaScript

JavaScript is a dynamically typed, interpreted scripting language. JavaScript uses `Array` as its primary linear data structure. Although it is called `Array`, data does not have to be stored side by side in memory, and it can hold heterogeneous elements together. Furthermore, `Arrays` are dynamic, allowing resizing.

- **What types are legal for subscripts?**
  In JavaScript, subscripts are actually treated as property keys, which are always either strings or symbols. When using bracket notation like `arr[index]`, JavaScript internally converts the subscript to a string if it's not already a string or a number. If the key is not found, JavaScript will not throw an error, instead it will return `undefined` and let the program continue.

  ```javascript
  console.log("1: Legal types for subscripts: ");
  let arr1 = [1, 2, 3, 4, 5];
  console.log(arr1[3]);
  let id = 3;
  console.log(arr1[id]);
  console.log(arr1["1"]);
  console.log(arr1[1.0]);
  console.log(arr1[[1]]);

  console.log(arr1[true]);
  console.log(arr1["ataturk"]);
  ```

  Printed Output:

  ```
  1: Legal types for subscripts:
  4
  4
  2
  2
  2
  undefined
  undefined
  ```

- **Are subscripting expressions in element references range checked?**
  No, JavaScript does not perform range checking for array indices during element access. If an out of bound value is given, JavaScript return `undefined`.

  ```javascript
  console.log("\n\n2: Range checking in element references:");
  console.log(arr1[10]);
  console.log(arr1[-1]);
  ```

  Printed Output:

  ```
  2: Range checking in element references:
  ```

```
undefined
undefined
```

- **Are ragged multidimensional arrays allowed?**
  Yes, Javascript allows any kind of multidimensional arrays.

```
console.log("\n\n3: Ragged multidimensional arrays:");

let ragged = [
    [1, 2, 3],
    [4, 5],
    [6, 7, 8, 9]
];
console.log(ragged);
console.log(Array.isArray(ragged));
console.log(Array.isArray(ragged[0]));
```

Printed Output:

```
3: Ragged multidimensional arrays:
[ [ 1, 2, 3 ], [ 4, 5 ], [ 6, 7, 8, 9 ] ]
true
true
```

- **Can array objects be initialized?**
  There are several ways to initialize array objects in JavaScript. You can initialize arrays by using a square bracket `[]` and giving a list of objects inside it. Arrays don't have to be of a single type, they can be mixed type. The array constructor, `Array(n)` does different initializations depending on the parameters it receives. If it takes a single element (= `n`), it creates an array of length `n` with uninitialized (empty) slots, but if it takes more than one list of elements it creates an array with those exact elements. To create an array with single-element, you can use `Array.of(x)` method. You can also give list of elements to this method. `Array.from()` converts iterable or array-like objects into real arrays. Also, typed arrays like `Uint32Array`, `Int16Array`, or `Float64Array` also provide a way to initialize arrays.

```
console.log("\n\n4: Initialization of arrays:");
let init1 = [1, 2, 3, "ataturk", true];
console.log(init1);

let init2 = Array(5); // one parameter -> 5 length with empty items.
console.log(init2);

let init3 = Array(3, 4, 5); // parameters are list
console.log(init3);

let init4 = Array.of(5); // creates [5]
```

```
console.log(init4);

let init5 = Array.from("ataturk"); // from iterable object
console.log(init5);

let init6 = Array.from({ length: 3 }, (_, i) => i + 1); //
console.log(init6);

let init7 = new Uint32Array(5);
console.log(init7);
```

Printed Output:

```
4: Initialization of arrays:
[ 1, 2, 3, 'ataturk', true ]
[ <5 empty items> ]
[ 3, 4, 5 ]
[ 5 ]
[
  'a', 't', 'a',
  't', 'u', 'r',
  'k'
]
[ 1, 2, 3 ]
Uint32Array(5) [ 0, 0, 0, 0, 0 ]
```

- **Are any kind of slices supported?**
  Yes. In JavaScript, array slicing can be done with the `slice()` method. When this method takes two parameters, it treats the first parameter as the start index and the second as the end index. If it takes only one parameter, it treats it as start and takes it to the end of the array. Although JavaScript does not allow negative indexing when subscripting, if a negative index is given as a index to `slice()` method it is allowed. When a negative index is given, -1 corresponds to the last element, and as the numbers decrease, going back to the beginning of array. Also, if no parameters are given to this function, it will slice from beginning to end, i.e. create a copy of the array.

  Additionally, JavaScript supports functional-style filtering using the `filter()` method, which can be used to create slices based on conditions rather than indices.

```
console.log("\n\n5: Array Slice:");
arr1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let slice = arr1.slice(1, 3); // (begin, end)
console.log(slice);
let slice2 = arr1.slice(3); // (begin)
console.log(slice2);
let slice3 = arr1.slice(-3); // (begin)
```

```
console.log(slice3);
let slice4 = arr1.slice(3, -2); // (begin, end)
console.log(slice4);
let slice5 = arr1.slice(-3, -1); // (begin, end)
console.log(slice5);
let slice6 = arr1.slice();
console.log(slice6); // copy

// using a filter function
let filtered = arr1.filter((item) => item > 5);
console.log("Filtered values (item > 5):", filtered);
```

Printed Output:

```
5: Array Slice:
[ 2, 3 ]
[
  4, 5,  6, 7,
  8, 9, 10
]
[ 8, 9, 10 ]
[ 4, 5, 6, 7, 8 ]
[ 8, 9 ]
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
Filtered values (item > 5): [ 6, 7, 8, 9, 10 ]
```

- **Which operators are provided?**
  In JavaScript, `==` and `===` are used for equality comparisons. The difference between them is that `==` provides type coercion (i.e., it tries to convert operands to the same type before comparison), while `===` performs strict comparison, checking both value and type. However, this distinction is not important for arrays since arrays are reference types. Therefore, if two variables point to the same place in memory (i.e. both reference to the same object) JavaScript returns `true`, otherwise returns `false`.

  In JavaScript, relational operators such as `<` and `>` do not provide meaningful comparison for arrays. Comparing arrays with these operators typically results in comparing their string representations.

  In JavaScript, `+` operator makes concatenation when applied to arrays, but not array concatenation like other languages. It applies type coercion first - converting the arrays to strings and then concatenates them as strings. Array concatenation can also be done using the `...`, spread operator. This operator works as it does in most languages. The `...` operator returns a new array by combining arrays side by side.

  JavaScript also supports the `in` operator with arrays, but it does not check whether a value exists in the array. Instead, it checks whether a specified index exists. For example, `4 in b` returns `false` since the array `b` doesn't have an element at index `1`, even if the number `4` exists as a value in the array.

```
console.log("\n\n6: Allowed Operations:");

const a = [1, 2, 3];
const b = [4, 5, 6];
const c = [1, 2, 3];
console.log("a:", a, "b:", b, "c:", c);
const d = a;

console.log("\nReference equality:");
console.log("a == c:", a == c);
console.log("a == d:", a == d);

// reference equality, same for arrays
console.log("a === c:", a === c);
console.log("a === d:", a === d);

console.log("\nComparison (not meaningful for arrays):");
console.log("a < c:", a < c);
console.log("a >= c:", a >= c);

console.log("\nConcatenation:"); // show using ... spread operator
const concat = [...a, ...b];
console.log("a + b (correct):", concat);
const wrongConcat = a + b;
console.log("a + b (wrong):", wrongConcat); // results in
    "1,2,34,5,6"     string

console.log("\n\"in\" operator (checks index):");
console.log("1 in a:", 1 in a);
console.log("10 in a:", 10 in a);
console.log("4 in b:", 4 in b);
```

Printed Output:

```
6: Allowed Operations:
a: [ 1, 2, 3 ] b: [ 4, 5, 6 ] c: [ 1, 2, 3 ]

Reference equality:
a == c: false
a == d: true
a === c: false
```

```
a === d: true

Comparison (not meaningful for arrays):
a < c: false
a >= c: true

Concatenation:
a + b (correct): [ 1, 2, 3, 4, 5, 6 ]
a + b (wrong): 1,2,34,5,6

"in" operator (checks index):
1 in a: true
10 in a: false
4 in b: false
```

1.4 Kotlin

Kotlin is a statically typed language that runs on the JVM. Kotlin supports `Arrays`, which are fixed-size data structures. They have to be homogeneous - while you can use a general type like `Any` to store different kinds of objects, the array still has a single, consistent type. Kotlin also provides dynamic and flexible data structures like `List` and `MutableList`, but for the purpose of this assignment, we focus on the fixed-size `Array` type.

Before we move on to the questions and their sample codes, it is necessary to explain a helper function used in the code, `printArray`. This function simply prints all the elements of an array side by side. The purpose of such a function is to make the example code shorter and more understandable.

```
fun printArray(arr: Any?) {
    when (arr) {
        is IntArray -> for (item in arr) print("$item ")
        is Array<*> -> for (item in arr) print("$item ")
        is Iterable<*> -> for (item in arr) print("$item ")
        }
    println()
}
```

- **What types are legal for subscripts?**
  In Kotlin, array subscripts must be integer. Only integer types can be used after an explicit conversion if they're not int by default. Other types are illegal to use, and gives compile time error.

```
println("1. Legal types for subscripts:")
val arr1 = arrayOf(1, 2, 3, 4, 5)
println(arr1[0])
// println(arr1[0.5]) Compile time error: argument type mismatch
// println(arr1["3"]) Compile time error: argument type mismatch
// println(arr1[true])  Compile time error: argument type mismatch
// println(arr1[0..2]) argument type mismatch: actual type is '
    IntRange', but 'Int' was expected.
/*
var ind : Long = 3
println(arr1[ind]) // Compile time error: argument type mismatch:
    actual type is 'Long', but 'Int' was expected.
*/
```

  Printed Output:

```
1. Legal types for subscripts:
1
```

- **Are subscripting expressions in element references range checked?**
  Yes. In Kotlin, subscripting expressions are range-checked, meaning accessing an index outside the valid range of an array will raise an error. Kotlin checks that in runtime.

```kotlin
println("\n2. Range checking in subscripting expressions")
// println(arr1[6]) // Runtime error: ArrayIndexOutOfBoundsException
// println(arr1[-1]) // Runtime error: ArrayIndexOutOfBoundsException
val num = 68
// println(arr1[num]) // Runtime error:
    ArrayIndexOutOfBoundsException.
```

Printed Output:

```
2. Range checking in subscripting expressions
```

- **Are ragged multidimensional arrays allowed?**
  Yes, Kotlin allows any kind of multidimensional arrays.

```kotlin
println("\n3. Ragged multidimensional arrays")
val raggedArr = arrayOf(arrayOf(1, 2, 3), arrayOf(4, 5), arrayOf(6,
    7, 8, 9))
for (i in raggedArr.indices) {
    printArray(raggedArr[i])
}
println("Type of raggedArr: ${raggedArr::class.simpleName}") // Array
    <Array<Int>>
println("Type of raggedArr[0]: ${raggedArr[0]::class.simpleName}") //
     Array<Int>
println("Type of raggedArr[0][0]: ${raggedArr[0][0]::class.simpleName
    }") // Int
```

Printed Output:

```
3. Ragged multidimensional arrays
1 2 3
4 5
6 7 8 9
Type of raggedArr: Array
Type of raggedArr[0]: Array
Type of raggedArr[0][0]: Int
```

- **Can array objects be initialized?**
  Yes, Kotlin provides several ways to initialize arrays. You can directly provide the elements using the `arrayOf()` function, use array constructors by specifying the size, or initialize arrays with custom logic using lambda expressions. Kotlin also supports creating arrays with default or null values.

```kotlin
println ("\n4. Initialization of arrays")

val arr2 = arrayOf(1, 2, 3, 4, 5, "Ataturk", 3.14, true) // mix type
```

```
printArray(arr2)

val intArr = IntArray(5) // size 5, 00000
printArray(intArr)

val lambdaArr = Array(5) { it * 2 }
printArray(lambdaArr) // 0 2 4 6 8

val arr4 = arrayOfNulls<Int>(5)
printArray(arr4)
println(arr4::class.simpleName)

val arr5 = Array<Int?>(5) { null }
printArray(arr5)
```

Printed Output:

```
4. Initialization of arrays
1 2 3 4 5 fdbjlka 3.14 true
0 0 0 0 0
0 2 4 6 8
null null null null null
Array
null null null null null
```

- **Are any kind of slices supported?**
  Yes. Kotlin supports array slicing. You can specify start and end parameters to determine the range when slicing. You can also specify step parameter for step slicing. Result of these slices will be `ArrayList`. If you want an array type slice, you can use the `copyOfRange` method to slice by copying a specific part of the array.

```
println("\n5. Array slices")

val arr6 = arrayOf(1, 2, 3, 4, 5)
val slice = arr6.slice(1..3) // (begin, end) inclusive
println("Kind of slice: ${slice::class.simpleName}")
printArray(slice)


val slice3 = arr6.slice(0..2 step 2) // step 2
println("Kind of slice: ${slice3::class.simpleName}")
printArray(slice3)

// To achive array when sliced:
val coppied = arr6.copyOfRange(1, 3) // (begin, end) exclusive
```

```
println("Kind of slice: ${coppied::class.simpleName}")
printArray(coppied)
```

Printed Output:

```
5. Array slices
Kind of slice: ArrayList
2 3 4
Kind of slice: ArrayList
1 3
Kind of slice: Array
2 3
```

- **Which operators are provided?**
  In Kotlin, one of the array operators provided is `contentEquals`. This checks whether two arrays have the same elements in the same order. Using the equality operator `==` in Kotlin calls the `equals()` function [9]. However, the function is not overridden for arrays, and thus it checks whether the two variables refer to the same array. As a result, even if two arrays contain the same values, `a == b` will return `false` unless they are the same object. For reference comparison, Kotlin uses the `===` operator, which checks whether two references point to the exact same object in memory. This is known as referential equality. In addition, Kotlin allows array concatenation using the `+` operator. Besides, it can also be added to the existing array using `+=` operator. Moreover, it is possible to check whether an element is in the array with the `in` operator. This operator returns `true` if the value exists, otherwise `false`.

```
println("\n6.Allowed Operations")
val a = arrayOf(1, 2, 3)
val b = arrayOf(4, 5, 6)
val c = arrayOf(1, 2, 3)
print("a: ")
printArray(a)
print("b: ")
printArray(b)
print("c: ")
printArray(c)

println("Compare contents:") // Checks if the two specified arrays
    are structurally equal to one another.
println("a contentEquals b: ${a contentEquals b}")
println("a contentEquals c: ${a contentEquals c}")

println("Structural equality:") // two objects have the same content
    or structure, but for arrays: assigned variables point to the same
     object.
println("a == b: ${a == b}")
println("a == c: ${a == c}")

println("Referential equality:") // two references pointing to the
    same object
println("a === b: ${a === b}")
println("a === c: ${a === c}")

val d = a
println("a == d: ${a == d}")
println("a === d: ${a === d}")

println("Concatenation:")
val concat = a + b
printArray(concat)

println("Contains:")
println("1 in a: ${1 in a}")
println("10 in a: ${10 in a}")
```

Printed Output:

```
6.Allowed Operations
a: 1 2 3
b: 4 5 6
c: 1 2 3
Compare contents:
a contentEquals b: false
a contentEquals c: true
Structural equality:
a == b: false
a == c: false
Referential equality:
a === b: false
a === c: false
a == d: true
a === d: true
Concatenation:
1 2 3 4 5 6
Contains:
1 in a: true
10 in a: false
```

1.5 PHP

PHP is a dynamically typed, interpreted scripting language. It offers a single, flexible `Array` data structure that can behave as both a numerically indexed array and as an associative map. Internally, PHP arrays are implemented as ordered hash tables, allowing mixed key types and dynamic resizing. For this reason, `array` elements are not stored contiguously in memory like in low-level languages.

- **What types are legal for subscripts?**

  PHP is different from most other languages: It tries to reduce the number of errors by automatically converting many types of subscripts into valid array keys. This includes strings. If a non-integer string is given, it gives an 'undefined character' warning. In this conversion, floating point numbers are converted with the round operation, and will give a 'Deprecated' warning (ie, this is deprecated, but we can still use it without error). Booleans are converted to 0 if false and to 1 if true. Given something that cannot be converted to an integer (like another array), PHP will give a runtime error.

  However, if you create them as a key-value pair like in a map, not as an array, then anything you define as a key (including null) can be accessed later with subscripting. Since the assignment asked for an array, this is not explained for this and further questions.

```
echo "1: Legal types for subscripts:\n";
$arr1 = array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
echo $arr1[3] . "\n";
$index = 3;
echo $arr1[$index] . "\n";
echo $arr1["a"] . "\n"; # Warning: Undefine array key "a"
echo $arr1["3"] . "\n";
echo $arr1[3.8] . "\n"; // Deprecated: Implicit conversion from float
    3.8 to int loses precision on line 19
$boolIndex = true;
echo $arr1[$boolIndex] . "\n"; // true => 1
$indArr = array(4);
# echo $arr1[$indArr] . "\n"; // Fatal error: Uncaught TypeError:
   Cannot access offset of type array on array
print_r($arr1);
```

Printed Output:

```
1: Legal types for subscripts:
4
4

Warning: Undefined array key "a" in C:\Users\Msı\CS_315\Homework1\Codes\22202709_Ozongun_
HalilArda.php on line 8

4
```

```
Deprecated: Implicit conversion from float 3.8 to int loses precision in C:\Users\Msı\CS_315\
Homework1\Codes\22202709_Ozongun_HalilArda.php on line 10
4
2
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 4
    [4] => 5
    [5] => 6
    [6] => 7
    [7] => 8
    [8] => 9
    [9] => 10
)
```

- **Are subscripting expressions in element references range checked?**
  Since PHP does not have indexes, range checking is not performed, but valid indexes must still be entered when accessing array elements. Accessing an invalid index does not crash the program, but triggers a warning and returns `NULL`.

```
echo "\n\n2: Range checking in element references:\n";
// echo $arr1[10]; // Warning: Undefined offset
// echo $arr1[-1]; // Warning: Undefined offset
$idx = 9;
// echo $arr1[$idx] . "\n"; // Warning: Undefined offset
```

Printed Output:

```
2: Range checking in element references:
```

- **Are ragged multidimensional arrays allowed?**
  Yes, PHP allows any kind of multidimensional arrays.

```
echo "\n\n 3: Ragged multidimensional arrays: ". "\n";
$ragged = array(
    array(1, 2, 3),
    array(4, 5),
    array(6, 7, 8, 9)
);
print_r($ragged);
```

Printed Output:

```
3: Ragged multidimensional arrays:
Array
(
    [0] => Array
        (
            [0] => 1
            [1] => 2
            [2] => 3
        )

    [1] => Array
        (
            [0] => 4
            [1] => 5
        )

    [2] => Array
        (
            [0] => 6
            [1] => 7
            [2] => 8
            [3] => 9
        )

)
```

- **Can array objects be initialized?**
  PHP supports some ways to initialize arrays: listing all elements, giving as a map, giving key indexes and a value for all, and range of values. Giving it as a map is not intuitive, but it is no different from any other methods, because in all cases PHP has to hold a key implicitly or explicitly.

```php
echo "\n\n4: Initialization of arrays: \n";
$standartInit = array(1, 3, 2);
$assocInit = array(0 => 1, 2 => 2, 1 => 3);
$shortSyntax = [1, 2, 3, 4, 5];
$defaultInit = array_fill(2, 5, 0); # start from 2. length 5. fill
    all with 0
$rangeInit = range(7, 5); # 7 to 5
print_r($defaultInit);
print_r($rangeInit);
echo "Arrays are ";
if ($standartInit == $assocInit) {
    echo "equal.\n";
} else {
    echo "not equal.\n";
}
```

Printed Output:

```
4: Initialization of arrays:
Array
(
    [2] => 0
    [3] => 0
    [4] => 0
    [5] => 0
    [6] => 0
)
Array
(
    [0] => 7
    [1] => 6
    [2] => 5
)
Arrays are equal.
```

- **Are any kind of slices supported?**
  In PHP, slicing is supported through a function: array_slice(). When slicing, you can give start index and length of slice as a parameter. You can use negative index for start in here. There is no step slicing like Python.

```
echo "\n\n5: Array Slice: \n";

$slice = array_slice($arr1, 1, 2); // (begin, length)
echo "Slice from index 1, length 2:\n";
print_r($slice);

$slice = array_slice($arr1, 1); // (begin)
echo "Slice from index 1 to end:\n";
print_r($slice);

$slice = array_slice($arr1, 0, 2);
echo "Slice from beginning, length 2:\n";
print_r($slice);

$slice = array_slice($arr1, -4, 2); # start from -4th index, length =
    2
echo "Slice from -4:\n";
print_r($slice);
```

Printed Output:

```
5: Array Slice:
Slice from index 1, length 2:
Array
(
    [0] => 2
    [1] => 3
)
Slice from index 1 to end:
Array
(
    [0] => 2
    [1] => 3
    [2] => 4
    [3] => 5
    [4] => 6
    [5] => 7
    [6] => 8
    [7] => 9
    [8] => 10
)
Slice from beginning, length 2:
Array
(
    [0] => 1
    [1] => 2
)
Slice from -4:
Array
(
    [0] => 7
    [1] => 8
)
```

- **Which operators are provided?**
  In PHP, the `==` operator checks if two arrays have the same key-value pairs. Here, the order of the keys is not important. If both arrays contain the same elements with the same keys, the result is `true`, even if the orders of keys are different [13]. On the other hand `===` operator, check for both value and key order (and type) [13]. If the arrays differ in key order, or if one is a reference and the other is not, `===` returns `false`.

  Relational operators like `<` and `<=` can also be used on arrays in PHP, but their behavior is less intuitive. These operators compare arrays based on the first element that differs when looping through them in order. This behavior should generally be avoided for comparing full arrays unless explicitly intended.

When using the `+` operator, PHP merges two arrays. If keys in the second array already exist in the first, they are ignored. This is a key-based merge, not a concatenation like in other languages. If the keys are unique, the arrays are merged side by side.

To check for the existence of a value in an array, PHP provides the `in_array()` function. It returns `true` if the value exists in the array, otherwise `false`.

```
echo "\n\n6: Allowed Operations:\n";

$a = array(1, 2, 3);
$b = array(4, 5, 6);
$c = array(1, 2, 3);
$onlyOrderDifferent = array(2 => 3, 0 => 1, 1 => 2);
$onlyKeyDifferent = array(1 => 1, 2 => 2, 3 => 3);
$onlyFirstValueBigger = array(2, -1, -1);

echo "a: "; print_r($a);
echo "b: "; print_r($b);
echo "c: "; print_r($c);
echo "onlyOrderDifferent: "; print_r($onlyOrderDifferent);
echo "onlyKeyDifferent: "; print_r($onlyKeyDifferent);
echo "onlyFirstValueBigger: "; print_r($onlyFirstValueBigger);

echo "\nCompare contents:\n";
echo "a == b: "; echo $a == $b ? "true\n" : "false\n";
echo "a == c: "; echo $a == $c ? "true\n" : "false\n";
echo "a != c: "; echo $a != $c ? "true\n" : "false\n";

echo "a == onlyOrderDifferent: ";
echo $a == $onlyOrderDifferent ? "true\n" : "false\n";

echo "a == onlyKeyDifferent: ";
echo $a == $onlyKeyDifferent ? "true\n" : "false\n";

echo "a == f (same values, string vs int): ";
$f = ['1', '2', '3'];
echo $a == $f ? "true\n" : "false\n";

echo "a < c: "; echo $a < $c ? "true\n" : "false\n";
echo "a <= c: "; echo $a <= $c ? "true\n" : "false\n";

echo "a < onlyFirstValueBigger: ";
echo $a < $onlyFirstValueBigger ? "true\n" : "false\n";
echo "\nCompare references (identity):\n";
echo "a === c: "; echo $a === $c ? "true\n" : "false\n";
$d = $a;
echo "a === d: "; echo $a === $d ? "true\n" : "false\n";
echo "a === onlyOrderDifferent (different keys/order): ";
echo $a === $onlyOrderDifferent ? "true\n" : "false\n";
echo "a === f (same values, string vs int): ";
echo $a === $f ? "true\n" : "false\n";

echo "\nConcatenation:\n";
$sameKeyDiffVal = $a + $b;
echo "a + b:\n"; print_r($sameKeyDiffVal);
$d = array(6 => 7, 8 => 9, 10 => 11);
echo "d: "; print_r($d);
$diffKey = $a + $d;
echo "a + d:\n"; print_r($diffKey);

echo "\nContains:\n";
echo "in_array(2, a): ";
echo in_array(2, $a) ? "true\n" : "false\n";
echo "in_array(10, a): ";
echo in_array(10, $a) ? "true\n" : "false\n";
```

Printed Output:

```
6: Allowed Operations:
a: Array
(
    [0] => 1
    [1] => 2
    [2] => 3
)
b: Array
(
    [0] => 4
    [1] => 5
    [2] => 6
)
c: Array
(
    [0] => 1
    [1] => 2
    [2] => 3
)
```

```
onlyOrderDifferent: Array
(
    [2] => 3
    [0] => 1
    [1] => 2
)
onlyKeyDifferent: Array
(
    [1] => 1
    [2] => 2
    [3] => 3
)
onlyFirstValueBigger: Array
(
    [0] => 2
    [1] => -1
    [2] => -1
)

Compare contents:
a == b: false
a == c: true
a != c: false
a == onlyOrderDifferent: true
a == onlyKeyDifferent: false
a == f (same values, string vs int): true
a < c: false
a <= c: true
a < onlyFirstValueBigger: true

Compare references (identity):
a === c: true
a === d: true
a === onlyOrderDifferent (different keys/order): false
a === f (same values, string vs int): false

Concatenation:
a + b:
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
)
d: Array
(
    [6] => 7
    [8] => 9
    [10] => 11
)
a + d:
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [6] => 7
    [8] => 9
    [10] => 11
)

Contains:
in_array(2, a): true
in_array(10, a): false
```

1.6 Python - NumPy

Python is a dynamically typed, interpreted programming language. Its built-in `list` is a dynamic and heterogeneous data structure. For this reason, we use the `ndarray` from the `NumPy` library in this assignment. `ndarray` elements have a fixed type, and the `NumPy` library supports a wide range of operations on them.

- **What types are legal for subscripts?**
  For Numpy ndarrays, array subscripts must be integer. Only integer types can be used after an explicit conversion if they're not int by default. Other types are illegal to use, and python gives error.

```
print("1: Legal types for subscripts:")
npArr1 = np.array([1, 2, 3, 4, 5])
print(npArr1[3])
# 64 bit integer
print(npArr1[np.int64(3)])
#print(npArr1[3.2]) # IndexError
#print(npArr1["2"]) # IndexError
#precNum = 3.3
#print(npArr1[precNum]) # IndexError
```

  Printed Output:

```
1: Legal types for subscripts:
4
4
```

- **Are subscripting expressions in element references range checked?**
  Yes. In Python, subscripting expressions are range-checked, meaning accessing an index outside the valid range of an array will raise an error. Unlike many other languages, Python supports negative indexing, where -1 refers to the last element, -2 to the second last, and so on. However, even negative indices are bounded, you can give indexes from -1 to -(length of array). If you give a smaller value, it will again raise an error.

```
print("2: Range checking in subscripting expressions:")
#print(npArr1[5]) # IndexError: index out of bounds
print(npArr1[-1])
#print(npArr1[-6]) # IndexError: index out of bounds
```

  Printed Output:

```
2: Range checking in subscripting expressions:
5
```

- **Are ragged multidimensional arrays allowed?**
  No, it is not allowed to create ragged multidimensional arrays. However, if you specify the type as `object`, NumPy will create an array of lists, making it possible to have varying lengths. Still, this is not a true NumPy array of arrays, so I consider ragged arrays as not allowed.

```python
print("3: Ragged multidimensional arrays:")
multiDimArr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(multiDimArr)
print("Type of multiDimArr:", type(multiDimArr))
print("Type of first element of multiDimArr:", type(multiDimArr[0]))
#raggedArr = np.array([[1, 2, 3], [4, 5], [7, 8, 9]]) # ValueError:
    The requested array has an inhomogeneous shape.
raggedArr = np.array([[1, 2, 3], [4, 5], [7, 8, 9]], dtype=object)
print(raggedArr)
print("Type of raggedArr:", type(raggedArr))
print("Type of first element of raggedArr:", type(raggedArr[0]))
```

Printed Output:

```
3: Ragged multidimensional arrays:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Type of multiDimArr: <class 'numpy.ndarray'>
Type of first element of multiDimArr: <class 'numpy.ndarray'>
[list([1, 2, 3]) list([4, 5]) list([7, 8, 9])]
Type of raggedArr: <class 'numpy.ndarray'>
Type of first element of raggedArr: <class 'list'>
```

- **Can array objects be initialized?**
  Yes, Numpy provides many different ways. You can give list of elements, or you can use some methods to fill the array. Below are a few examples:

```python
print("4: Initialization of array objects:")
print(np.zeros(5))
print(np.ones(5))
print(np.empty(5))
print(np.full(5, 5))
print(np.arange(5))
print(np.linspace(0, 10, 5))
print(np.random.rand(5))
print(np.random.randn(5))
print(np.random.randint(5))
print(np.random.random(5))
print(np.random.choice(5, 5))
print(np.random.permutation(5))
print(np.random.shuffle(np.arange(5)))
print(np.random.seed(5))
a = np.array([1, 2, 3, 4, 5])
```

```
c = np.array([2])*5
print("Array 1:", a, len(a))
print("Array 2:", c, len(c))
```

Printed Output:

```
4: Initialization of array objects:
[0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1.]
[5 5 5 5 5]
[0 1 2 3 4]
[ 0.   2.5  5.   7.5 10. ]
[0.28508668 0.49099169 0.06152563 0.04973992 0.50949782]
[-0.85974521  0.80750307 -0.58291767  0.2364831  -0.45395473]
2
[0.37768406 0.52512864 0.4268821  0.64492011 0.81168118]
[4 1 1 4 2]
[0 1 3 4 2]
None
None
Array 1: [1 2 3 4 5] 5
Array 2: [10] 1
```

- **Are any kind of slices supported?**
  Yes. Array slicing is supported in Python. Slicing can be done specifying the start and end indices, and different from other languages step slicing is provided(skips elements based on the step). In addition to standard slicing, Python also supports filtering using list comprehensions, allowing you to create arrays considering some conditions. The `...` (ellipsis) in NumPy is used to represent all unspecified dimensions when indexing, which is useful for high-dimensional arrays.

```
print("5: Array slicing:")
b = a[1:4]
print("begin and end given", b)
print(type(b))
b = a[:2]
print("only end given", b)
b = a[2:]
print("only begin given", b)
b = a[:]
print("only colon given", b)
b = a[1:4:2]
print("Step given", b)
b = a[::-1]
print("Negative step given", b)
```

```
arr = [1, 2, 3, 4, 5, 6]
filtered = [x for x in arr if x % 3 > 1]
print("Filtered array:", filtered)
multiDim = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print ("First column:", multiDim[..., 0])
```

Printed Output:

```
5: Array slicing:
begin and end given [2 3 4]
<class 'numpy.ndarray'>
only end given [1 2]
only begin given [3 4 5]
only colon given [1 2 3 4 5]
Step given [2 4]
Negative step given [5 4 3 2 1]
Filtered array: [2, 5]
First column: [1 4 7]
```

- **Which operators are provided?**

  When comparing arrays, the `==` operator compares arrays element-wise. Unlike most languages, instead of returning a single boolean value, it returns a boolean array whose size is the same as the size of the arrays, and each element in the result array is the result of comparing the order of the elements in the operand arrays. Likewise, relational operators such as `<`, `>`, `<=`, and `>=` are also applied elementally and return a Boolean array for the result. The size of the arrays must be the same, otherwise Python will give an error.

  If exact memory equality (reference identity) is needed, the `is` operator can be used. This checks whether two variables point to the exact same object in memory.

  Python supports a wide range of arithmetic operators for arrays, including `+`, `-`, `*`, `/`, `//`, `%`, `**`, and more. All of them are applied element-wise and return an array as a result. Bitwise operations such as `&`, `|`, `^`, `~`, `<<`, and `>>` also behave element-wise. You can also add a constant to all elements in an array, just like adding an integer to a number. You can also do other arithmetic operations with a constant.

  Python also has an operator to calculate the dot product of two arrays, `@`. Membership testing can be done using the `in` operator. If value exists in array, Python returns `True`, otherwise `False`. You can also reverse an array, using slicing. Also, since slicing and using negative numbers for step are legal in Python, you can reverse an array using `a[::-1]` syntax. There are other methods for reversing, but they are not included because they are functions.

```
print("\n\n6: Operators provided:")
a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
b = np.array([11, 12, 13, 14, 15, 16, 17, 18, 19])
c = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
print("Array a:", a)
print("Array b:", b)
print("Array c:", c)
shorter = np.array([1, 2, 3])

print("Equality Checking:")
print("a == c:", a == c)
print("a == b:", a == b)
# print("a == shorter:", a == shorter) # ValueError: operands could
    not be broadcast together with shapes (9,) (3,)


print("Comparison:")
print(" a < c:", a < c)
print(" a < b:", a < b)
print(" a > c:", a > c)
print(" a <= c:", a <= c)

print("Memory equality:")
print("a is c:", a is c)
print("a is a:", a is a)
d = a
print("a is d:", a is d)

print("Adding constant = 10000 to all elements:", a + 10000)

print("Arithmetic operations using a and c arrays:")
print("a + c:", a + c)
print("a - c:", a - c)
print("a * c:", a * c)
print("a / c:", a / c)
print("a % c:", a % c)
print("a ** c:", a ** c)
print("a // c:", a // c)

print("Bitwise operations using a and c arrays:")
print("a & c:", a & c)
print("a | c:", a | c)
print("a ^ c:", a ^ c)
print("~a:", ~a)
print("a << 2:", a << 2)
print("a >> 2:", a >> 2)

print("Dot product of a and c:", a @ c)
```

```python
print("Contains check")
print("2 in a:", 2 in a)
print("10 in a:", 10 in a)

print("Reversing an array (a) ", a[::-1])
```

Printed Output:

```
6: Operators provided:
Array a: [1 2 3 4 5 6 7 8 9]
Array b: [11 12 13 14 15 16 17 18 19]
Array c: [1 2 3 4 5 6 7 8 9]
Equality Checking:
a == c: [ True  True  True  True  True  True  True  True  True]
a == b: [False False False False False False False False False]
Comparison:
 a < c: [False False False False False False False False False]
 a < b: [ True  True  True  True  True  True  True  True  True]
 a > c: [False False False False False False False False False]
 a <= c: [ True  True  True  True  True  True  True  True  True]
Memory equality:
a is c: False
a is a: True
a is d: True
Adding constant = 10000 to all elements: [10001 10002 10003 10004 10005 10006 10007 10008 10009]
Arithmetic operations using a and c arrays:
a + c: [ 2  4  6  8 10 12 14 16 18]
a - c: [0 0 0 0 0 0 0 0 0]
a * c: [ 1  4  9 16 25 36 49 64 81]
a / c: [1. 1. 1. 1. 1. 1. 1. 1. 1.]
a % c: [0 0 0 0 0 0 0 0 0]
a ** c: [        1         4        27       256      3125     46656    823543
  16777216 387420489]
a // c: [1 1 1 1 1 1 1 1 1]
Bitwise operations using a and c arrays:
a & c: [1 2 3 4 5 6 7 8 9]
a | c: [1 2 3 4 5 6 7 8 9]
a ^ c: [0 0 0 0 0 0 0 0 0]
~a: [ -2  -3  -4  -5  -6  -7  -8  -9 -10]
a << 2: [ 4  8 12 16 20 24 28 32 36]
a >> 2: [0 0 0 1 1 1 1 2 2]
Dot product of a and c: 285
Contains check
```

```
2 in a: True
10 in a: False
Reversing an array (a)  [9 8 7 6 5 4 3 2 1]
```

1.7 Rust

Rust is a statically typed, compiled systems programming language. Rust supports fixed-size `array` types, which store elements of a single type. Rust also provides `Vec<T>` type for dynamic collections, but we will stick to the `arrays`. Also Rust enforces strict memory safety, including bounds checking on array accesses, ensuring robust and predictable behavior.

Before we move on to the questions and their sample codes, it is necessary to explain helper functions used in the code, `print_array` and `type_of`. These helper functions make output more readable, and easy to understand for someone who doesn't know Rust. Function `type_of` takes a reference to any value and returns a string representing its type name. Function `print_array` takes an array of integers (&[i32]) and prints its contents, one element at a time, separated by spaces.

```
fn type_of<T>(_: &T) -> &'static str {
    type_name::<T>()
}

fn print_array(arr: &[i32]) {
    for i in 0..arr.len() {
        print!("{}  ", arr[i]);
    }
    println!(" ");
}
```

- **What types are legal for subscripts?**
  In Rust, only `usize` can be used as a subscript index. Other types such as `i32` are not supported. `usize` is an unsigned integer, and size is determined by architecture (32-bit on 32-bit systems, 64-bit on 64-bit systems).

```
println!("1: Legal types for subscripts:");
let arr1 = [1, 2, 3, 4, 5];
println!("{}", arr1[0]);
// println!("{}", arr1["0"]); // Compile time error: error[E0277]:
   the type `[{integer}]` cannot be indexed by `&str`
//println!("{}", arr1[0.0]); // Compile time error: error[E0277]: the
    type `[{integer}]` cannot be indexed by `float`
// println!("{}", arr1[0.0f32]); // Compile time error: error[E0277]:
    the type `[{integer}]` cannot be indexed by `f32`
let mut i = 0;
println!("{}", arr1[i]);
println!("type: of i: {}", type_of(&i));
let i2: u32 = 0;
// println!("{}", arr1[i2]); //Compile time errorL error[E0277]: the
   type `[{integer}]` cannot be indexed by `u32`
```

Printed Output:

```
1: Legal types for subscripts:
1
1
type: of i: usize
```

- **Are subscripting expressions in element references range checked?**
Yes. Rust performs bounds checking. If the index is a constant, it is checked at compile time and will result in a compile-time error if out of bounds. If it is a variable, it will be checked at runtime.

```
println!("\n\n2: Range checking in Subscripting expressions: ");
//println!("{}", arr1[9]);  // compile-time denial -> Don't compile.
//println!("{}", arr1[-3]); // Compile time error. Negative integers
    cannot be used to index on a [{integer},5]
i = 98;
// println!("{}", arr1[i]); // Run-time error.
```

Printed Output:

```
2: Range checking in Subscripting expressions:
```

- **Are ragged multidimensional arrays allowed?**
No, Rust doesn't allow ragged multidimensional arrays. If you create an array of arrays with different lengths it will result in a compile time error. However, you can create array of vectors (or some other datatypes), which can have different size.

```
println!("\n\n3: Ragged multidimensional arrays: ");
let multidimensionalArr = [[1, 2, 3], [4, 5, 6]];
println!("type: of multidimensionalArr: {}", type_of(&
    multidimensionalArr));

//let raggedArr = [[1, 2, 3], [4, 5]]; // Compile time error.
//println!("type: of raggedArr: {}", type_of(&raggedArr));

// array of vectors
let arr_of_vec = [vec![1, 2, 3], vec![4, 5, 6]];
println!("type: of arr_of_vec: {}", type_of(&arr_of_vec));
```

Printed Output:

```
3: Ragged multidimensional arrays:
type: of multidimensionalArr: [[i32; 3]; 2]
type: of arr_of_vec: [alloc::vec::Vec<i32>; 2]
```

- **Can array objects be initialized?**
Yes. Like most other programing languages, Rust allows arrays to be initialized in different ways. You can list the elements directly (and declare it mutable or not), or you can initalize with same value with

length, or you can convert from other data structures (like vectors). Essential point is, the size must be known, either by explicitly specifying it or by providing a fixed number of elements.

```
println!("\n\n4: Initialization of arrays: ");
let mut arr2 = [5, 6, 7, 8, 9]; // Mutable array
let arr3 = [5, 6, 7, 8, 9]; // Immutable array
let arr4 = [6; 5];
print_array(&arr4);
let arr6: [i32; 5] = [45263, 45263, 45263, 45263, 45263];
print_array(&arr6);
let vec = vec![43, 45, 67, 89, 90]; // Vector
let arr7: [i32; 5] = vec.try_into().unwrap();
print_array(&arr7);
```

Printed Output:

```
4: Initialization of arrays:
6  6  6  6  6
45263  45263  45263  45263  45263
43  45  67  89  90
```

- **Are any kind of slices supported?**
  Yes, Rust supports slicing arrays, but the result is not an array; it is a slice, which is its own type. When slicing, you can use ranges and you don't need to tell begin or end index (you can tell both, or one, or none). You can't do step slicing like in Python.

```
println!("\n\n5: Array Slices: ");
println!("\n\n5: Array Slices: ");
let slice = &arr1[1..4]; // 1 to 3

println!("slice: {:?}", slice);
println!("type of slice: {}", type_of(&slice));

let slice2 = &arr1[1..];
println!("slice2: {:?}", slice2);
println!("type of slice2: {}", type_of(&slice2));
// step slicing is not supported
```

Printed Output:

```
5: Array Slices:
slice: [2, 3, 4]
type of slice: &[i32]
slice2: [2, 3, 4, 5]
type of slice2: &[i32]
```

- **Which operators are provided?**
  In Rust, arrays don't have many operator options, but they do allow us to make comparisons about the elements of the array. The == operator returns `true` when the length, and respectively the elements, of two arrays are equal. On the other hand, Rust also allows comparisons of greater than and less than. When Rust compares arrays, <, >, <=, and >= again compare arrays lexiographically. For the comparison to == operator, it is worth to mention `std::ptr::eq()` function. This function returns `true` only if both references refer to the exact same object in memory. In the example, even though `a` and `c` have the same contents, they are different objects, so `eq(&a, &c)` returns `false`.

```
println!("\n\n6: Allowed Operations: ");

let a = [1, 2, 3];
let b = [4, 5, 6];
let c = [1, 2, 3];
println!("a: {:?}, b: {:?}, c: {:?}", a, b, c);

println!("Compare contents:");
println!("a == b: {}", a == b);
println!("a == c: {}", a == c);
println!("a != c: {}", a != c);
println!("a < b: {}", a < b);

println!("Compare references:");
println!("eq(&a, &b): {}", eq(&a, &b));
println!("eq(&a, &c): {}", eq(&a, &c));
let d = &a;
println!("eq(&a, d): {}", eq(&a, d));
```

Printed Output:

```
6: Allowed Operations:
a: [1, 2, 3], b: [4, 5, 6], c: [1, 2, 3]
Compare contents:
a == b: false
a == c: true
a != c: false
a < b: true
Compare references:
eq(&a, &b): false
eq(&a, &c): false
eq(&a, d): true
```

2 Evaluation

This section evaluates the programming languages in terms of readability and writability of array operations.

2.1 Dart

The array structures in `Dart` make the language easy to read and write, especially since they are very similar to the structures in classical scripting languages. Operations like subscripting and printing the array are very similar to other languages. The use of multidimensional arrays makes the language easier to write in places where such things are required, but it can make it difficult to read. Providing different methods for initializing the array, or slicing the array, increases the flexibility of the language, which in turn increases its writability. However, the need to distinguish between `growable` and `fixed-length` lists make it hard to read.

2.2 Go

Creating an array in `Go` is not very difficult syntactically, but it is almost the same as creating a `slice`, which I think makes the language difficult to understand. When you're writing, sometimes you don't know what you're creating, and when you're reading, you're not sure which variable you have. Other than that, I think there aren't any other readability and writeability issues with array structures, it's easy to get used to because it's similar to other languages.

2.3 JavaScript

`JavaScript` offers high flexibility with dynamic arrays. For arrays its syntax is familiar and easy to write. It allows more types for subscripting, thus when writing code it is easier to write compared to other languages. It also allows different types in same array, which makes it more writable.

2.4 Kotlin

Creating arrays in `Kotlin` has low writeability because it is done with constructors, but accessing elements etc. is very similar to other languages, so it is a good language in terms of readability and writeability in general. The fact that we have to specify types when creating arrays has a bad effect on writeability, but increases the reliability of the language.

2.5 PHP

`PHP` allows flexible list behavior via its `array` structure, which combines characteristics of arrays and hash maps. Although it offers various built-in functions for array manipulation, inconsistencies in naming and parameter order across functions can reduce readability. Dynamic typing can also introduce subtle bugs in list operations.

2.6 Python (NumPy)

`Python` may be the easiest language to write and read of all these languages. Creating an array is similar to most other languages. You can create arrays in multiple ways and when you access the indexies of the arrays, it performs range checking, but it also allows `negative indexing` which makes it easy to write. Because of the library support, you can create arrays in many ways. And since it allows `step slicing` or `list comprehension`, it has a lot of flexibility. It is also an easy language to write because it is similar to English. It also allows much more operations in terms of array operations compared to other languages.

2.7 Rust

I think `Rust` is the most difficult language to write and read among all these languages. Because you can only give `usize`-type integers for the index when subscripting. Also, the way `Rust` creates arrays is different from other languages. The syntax for slicing is also different and difficult than other languages. Although the operations it releases are not bad, it is generally a difficult language to syntax. This reduces its readability and writeability, but on the other hand some restrictions increase the reliability of the language.

2.8 Conclusion:

In my opinion, **NumPy ndarray** is much better than the other arrays in terms of both readability and writeability. The fact that it allows `negative indexing` unlike other languages, and especially the operator facilities it provides, definitely makes it easier to write code and puts it ahead.

3 Learning Strategy

This section describes the resources I used to do the assignment, and the process of doing the assignment.

3.1 Materials and Resources

I gathered information primarily from the official documentation of each language. Below is the list of official documentation pages I consulted for array-related syntax and semantics:

- Dart [1]
- Go [6]
- JavaScript [7]
- Kotlin [9]
- PHP [13]
- Python [11, 15]
- Rust [17]

In addition to the official documentation, I frequently used the following online platforms to understand nuances and resolve specific implementation challenges:

- **W3Schools** [19]: Thanks to the short but complete explanations it provides, it helped me a lot in the first learning of PHP and Javascript languages.
- **GeeksforGeeks** [3]: For detailed explanations, tutorials, and comparisons of array features across languages.
- **GitHub**: [4] For checking example projects, code snippets, and best practices in idiomatic array usage.
- **Stack Overflow** [18]: For clarifications on language-specific behaviors and edge cases, as well as resolving bugs and errors during implementation.

3.2 Experimental Setup and Process

I installed and configured each language's compiler or interpreter on my local machine using the official distribution sources [2, 5, 8, 10, 12, 14, 16]. For JavaScript, I used Node.js to execute scripts locally. For all languages, I accessed the documentation in my browser and practiced writing full source files to test each array-related feature as required.

After, I wrote source files addressing the questions mentioned. Each file was then compiled and/or interpreted depending on the language. I observed the output, analyzed runtime behavior, and noted differences in compile-time versus runtime error handling. This hands-on experimentation played a critical role in solidifying my understanding.

Several iterations were performed per language to ensure originality and correctness, especially to avoid replicating any existing online examples. I also ensured that my sample programs were complete and self-contained.

3.3 Personal Communication

When I was doing the assignment, they asked us to create HTML files for both JavaScript and PHP on the assignment page. Since I couldn't zip them together, I discussed this with Halil Altay Güvenir, the instructor of the course. I also got help from Mehmet Can Şakiroğlu, the teaching assistant of the course, to clarify some questions during the assignment. I would like to thank them both for their guidance.

References

[1] Dart documentation. `https://dart.dev/`. Accessed: 2025-04-01.

[2] Dart sdk. `https://dart.dev/get-dart`. Accessed: 2025-04-01.

[3] Geeksforgeeks. `https://www.geeksforgeeks.org`. Accessed: 2025-04-01.

[4] Github. `https://github.com`. Accessed: 2025-04-01.

[5] Go compiler. `https://go.dev/dl/`. Accessed: 2025-04-01.

[6] Go documentation. `https://go.dev/`. Accessed: 2025-04-01.

[7] Javascript documentation (mdn). `https://developer.mozilla.org/`. Accessed: 2025-04-01.

[8] Kotlin compiler. `https://kotlinlang.org/docs/command-line.html`. Accessed: 2025-04-01.

[9] Kotlin documentation. `https://kotlinlang.org/`. Accessed: 2025-04-01.

[10] Node.js. `https://nodejs.org/`. Accessed: 2025-04-01.

[11] Numpy documentation. `https://numpy.org/doc/`. Accessed: 2025-04-01.

[12] Php. `https://www.php.net/downloads`. Accessed: 2025-04-01.

[13] Php documentation. `https://www.php.net/manual/en/`. Accessed: 2025-04-01.

[14] Python. `https://www.python.org/downloads/`. Accessed: 2025-04-01.

[15] Python documentation. `https://docs.python.org/`. Accessed: 2025-04-01.

[16] Rust. `https://www.rust-lang.org/tools/install`. Accessed: 2025-04-01.

[17] Rust documentation. `https://doc.rust-lang.org/`. Accessed: 2025-04-01.

[18] Stack overflow. `https://stackoverflow.com`. Accessed: 2025-04-01.

[19] W3schools. `https://www.w3schools.com`. Accessed: 2025-04-01.