CS315 - Homework 2 Report

Halil Arda Özongun

ID: 22202709

Department of Computer Engineering

Bilkent University

Ankara, Turkey

April 27, 2025

1 Operations in each language

1.1 Dart

In Dart, multiple-way selection can be done in some different ways. The first one, and the one we will use in these assignments, is the `switch-case` statement, which is the most preferred way. Also in Dart, the `if-else if-else` statements and `maps` allows to do multiple-way statements.

- **What is the form and type of the control expression?**
  In Dart, control expression supports literals, variables, expressions, lists etc. Dart doesn't supports direct object comparison, instead their properties might be used.

```
print("1 - What is the form and type of the control expression?");

// variable
int value = 5;
switch (value) {
case 1:
  print("Control expression is an integer");
  break;
case 5:
  print("Control expression is an integer");
  break;
}

// expression
int val1 = 3;
int val2 = 4;
switch (val1 + val2) {
case 7:
  print("Control expression is an expression resulting in 7");
  break;
default:
  break;
}

// literal
switch (10) {
case 10:
  print("Control expression is a literal");
  break;
case 20:
  print("Control expression is a literal");
  break;
}
```

```
// string
String fruit = "apple";
switch (fruit) {
case "banana":
  print("Control expression is a string: banana");
  break;
case "apple":
  print("Control expression is a string: apple");
  break;
default:
  print("Unknown fruit");
}

// object
var obj = OBJ(5);
switch (obj.x) {
case 1:
  print("Control expression is an object property: 1");
  break;
case 5:
  print("Control expression is an object property: 5");
  break;
default:
  print("Unknown object property value: ${obj.x}");
}

// list
var expList = [1, 2, 3];
switch (expList) {
case [1, 2, 3]:
  print("Control expression is a list: $expList");
  break;
case [4, 5, 6]:
  print("Control expression is a list: $expList");
  break;
default:
  print("Unknown list: $expList");
}
```

Printed Output:

```
1 - What is the form and type of the control expression?
Control expression is an integer
Control expression is an expression resulting in 7
Control expression is a literal
Control expression is a string: apple
Control expression is an object property: 5
Control expression is a list: [1, 2, 3]
```

- **How are the selectable segments specified?**
  In Dart, selectable segments are defined using the case keyword, followed by a constant value and a colon :, then one or more actions. Actions can be single or multiple statements. If multiple cases share the same actions, they can be stacked without code between them. Optionally, curly braces can be used to group statements inside a case.

  ```
  print("\n\n2 - How are the selectable segments specified?");
  value = 12;
  switch (value) {
  case 5:
    print("Case 5: Single statement");
    break;
  case 10:
    print("Case 10: First statement");
    print("Case 10: Second statement");
    break;
  case 12:
  case 15:{
    print("Case 15: First statement");
    print("Case 15: Second statement");
    break;
  }

  default:
    break;
  }
  ```

  Printed Output:

  ```
  2 - How are the selectable segments specified?
  Case 15: First statement
  Case 15: Second statement
  ```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  In Dart this function works a little differently. If there is no statement in a case, the case flows to the following lines. However, if you don't want it to flow, you can exit it with the keyword `break`. But if there is already any statement for a case, the case will exit at the end of the case block, using break here does not change anything.

```
print("\n\n3 - Is execution flow restricted to just a single
    selectable segment?");
var l = [0, 1, 2, 4];
for (var item in l) {
print("Switching on item: $item");
switch (item) {
  case 0:
    break;
  case 1:
  case 2:
    print("case 2");
  case 4:
    print("case 4");
    break;
  case 8:
    print("case 8");
    break;
  default:
    print("Default case");
}
}
```

Printed Output:

```
3 - Is execution flow restricted to just a single selectable segment?
Switching on item: 0
Switching on item: 1
case 2
Switching on item: 2
case 2
Switching on item: 4
case 4
```

- **How are case values specified?**
  In Dart, case values must be constant expressions and must be known at compile time. Case values can be literals like integers, strings, or boolean values. Multiple case labels can be stacked without a break between them to match the same set of actions. If the values are not constant, Dart will raise an error. Dart also supports lists, objects etc. using pattern matching.

```
print("\n\n4 - How are case values specified?");

value = 3;
// literal
switch (value) {
```

```
case 1:
  print("Matched 1");
case 2:
  print("Matched 2");
}

// Multiple cases
print("\nMultiple case values:");
switch (value) {
case 1:
case 2:
case 3:
  print("Matched 1, 2, or 3");
default:
  print("No match");
}

String player = "Barella";
switch (player) {
case "Barella":
  print("Player is Barella");
  break;
case "Hakan":
  print("Player is Hakan");
  break;
default:
  print("Unknown player");
}

int caseValue = 2;
const int caseValue2 = 3;
switch (value) {
/*case caseValue:
  print("Matched variable case value: $caseValue");
  break;*/ // Error
case caseValue2:
  print("Matched constant case value: $caseValue2");
  break;
}

// object
var obj1 = OBJ(2);
switch (obj1.x) {
```

```
case 1:
  print("Matched object property value: ${obj1.x}");
  break;
case 2:
  print("Matched object property value: ${obj1.x}");
  break;
default:
  print("No match for object property value: ${obj1.x}");
}

// list
var list = [1, 2, 3];
switch (list) {
case [1, 2, 3]:
  print("Matched list: $list");
  break;
case [4, 5, 6]:
  print("Matched another list: $list");
  break;
default:
  print("No match for list: $list");
}
```

Printed Output:

```
4 - How are case values specified?

Multiple case values:
Matched 1, 2, or 3
Player is Barella
Matched constant case value: 3
Matched object property value: 2
Matched list: [1, 2, 3]
```

- **What is done about unrepresented expression values?**
  In Dart, if none of the `case` values are matched, `default` clause might be used to catch these unrepresented expression values.

```
  print("\n\n5 - What is done about unrepresented expression values?"
     );
  value = 100;

  // Without default case
  switch (value) {
    case 10:
      print("Matched 10");
  }
  // With default case
  print("\nWith default case:");
  switch (value) {
    case 10:
      print("Matched 10");
    default:
      print("No match for the given value: $value");
  }
```

Printed Output:

```
5 - What is done about unrepresented expression values?
No match for the given value: 100
```

1.2 Go

In Go, multiple-way selection can be done in several different ways. The first one, and the one we will use in these assignment, is the `switch-case` statement, which is the method usually prefered in Go. This `switch-case`, doesn't require a constant expression and also it can switch on any type, not just integers. Also in Go, the `if-else if-else` statement can be used for multiple way selection. Similar to Python, the `map` data structure can be used for multiple-way selection too. Go also has a `type switch`, where the `case` to be entered can change depending on the type of an interface at runtime.

- **What is the form and type of the control expression?**
  In Go, control expression can be any type of expression, including literals, variables, arithmetic expressions, etc. Also, type of the control expression can be any valid Go type — such as int, string, float64, etc.
  Go also supports type switches, where the control expression is an interface and cases are matched against the actual dynamic type of the variable. This allows more advanced control flow based on types rather than values.

```go
fmt.Println("\n1 - What is the form and type of the control
    expression?")

// Control expression as a variable
value := 5
switch value {
case 1:
    fmt.Println("Control expression is an integer")
case 5:
    fmt.Println("Control expression is an integer")
}

// Control expression as an expression
val1 := 3
val2 := 4
switch val1 + val2 {
case 7:
    fmt.Println("Control expression is an expression resulting in 7")
}

// Control expression as a literal
switch 10 {
    case 10:
    fmt.Println("Control expression is a literal")
    case 20:
    fmt.Println("Control expression is a literal")

}
```

```
// Control expression as a object
obj := Obj{4}
switch obj {
    case Obj{3}:
        fmt.Println("Object with x = 3")
    case Obj{4}:
        fmt.Println("Object with x = 4")
}

// Type switch
var i interface{}
items := []interface{}{1, "apple", [2]int{3, 4}, []int{1, 2, 3}, map[
   string]int{"key": 1}, true, nil, Obj{3}}

for _, i = range items {
    // print value and type of i
    fmt.Printf("Value: %v, Type: %T\t\t", i, i)
    switch v := i.(type) {
    case int:
        fmt.Println("Control expression is an integer")
    case string:
        fmt.Println("Control expression is a string")
    case [2]int:
        fmt.Println("Control expression is an array length 2")
    case []int:
        fmt.Println("Control expression is a slice")
    case map[string]int:
        fmt.Println("Control expression is a map")
    case bool:
        fmt.Println("Control expression is a boolean")
    case nil:
        fmt.Println("Control expression is nil")
    case Obj:
        if v.x == 3 {
            fmt.Println("Control expression is an Obj with x = 3")
        } else {
            fmt.Println("Control expression is an Obj")
        }
    default:
        fmt.Println("No match found")
    }
}
```

Printed Output:

```
1 - What is the form and type of the control expression?
Control expression is an integer
Control expression is an expression resulting in 7
Control expression is a literal
Object with x = 4
Value: 1, Type: int  Control expression is an integer
Value: apple, Type: string  Control expression is a string
Value: [3 4], Type: [2]int  Control expression is an array length 2
Value: [1 2 3], Type: []int  Control expression is a slice
Value: map[key:1], Type: map[string]int  Control expression is a map
Value: true, Type: bool  Control expression is a boolean
Value: <nil>, Type: <nil>  Control expression is nil
Value: {3}, Type: main.Obj  Control expression is an Obj with x = 3
```

- **How are the selectable segments specified?**
  Each case can be followed by a statement block, or can be leave as empty. Statements continue until the next case or until } if last case is executed. It executes only if the case pattern is matched and Go control flow enters that block.

```
fmt.Println("\n\n2 - How are the selectable segments specified?")
value = 5
switch value {
case 5:
    fmt.Println("Case 5: Single statement")
case 10:
    fmt.Println("Case 10: Multiple statements")
    fmt.Println("You can have more than one line in a case block")
}
```

Printed Output:

```
2 - How are the selectable segments specified?
Case 5: Single statement
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  Yes, in Go the first equal `case` blog is executed, other cases below it are not executed, so there is no fall through execution. But you can achieve fall through using `fallthrough` keyword. Also, same value cannot be in multiple cases.

```
fmt.Println("\n\n3 - Is execution flow restricted to just a single
    selectable segment?")
value = 6
switch value {
```

```
case 2, 4, 6:
    fmt.Println("Matched even number")
case 3, 9:
    fmt.Println("Matched divisible by 3")
default:
    fmt.Println("No match")
}
value = 6
switch value {
case 2, 4, 6:
    fmt.Println("Matched even number")
    fallthrough
case 3, 9:
    fmt.Println("Matched divisible by 3")
default:
    fmt.Println("No match")
}
/*switch value {
case 2, 4, 6:
    fmt.Println("Matched even number")
    case 3, 6, 9:
    fmt.Println("Matched divisible by 3")
}*/
```

Printed Output:

```
3 - Is execution flow restricted to just a single selectable segment?
Matched even number
Matched even number
Matched divisible by 3
```

- **How are case values specified?**
  In Go, `case` values can be constants, constant expressions, or types (in a type switch). However, the values used in case clauses must follow strict typing. It is required that all case values to be of the same type as the control expression. Constant literals like integers, strings, booleans, etc., can be used directly in case clauses. Also, Go allows arithmetic or constant expressions. Also, it is allowed to combining multiple values in a single case using commas. Also, in type switch, `case` values might be types.

```
fmt.Println("\n\n4 - How are case values specified?")

value = 3
// literal values
switch value {
case 1:
    fmt.Println("Matched 1")
case 2:
    fmt.Println("Matched 2")
case 3:
    fmt.Println("Matched 3")
}

// expressions
switch value {
case 2 + 1:
    fmt.Println("Matched 3")
case 10 - 3:
    fmt.Println("Matched 7")
}

// expression with variables
var m, n int = 2, 3
switch value {
case m * n:
    fmt.Println("Matched 6")
case m + n:
    fmt.Println("Matched 5")
}

// multiple case values

switch value {
case 1, 2, 3:
    fmt.Println("Matched 1, 2, or 3")
case 4, 5, 6:
    fmt.Println("Matched 4, 5, or 6")
}

// variables
var a, b int = 1, 3
switch value {
case a:
    fmt.Println("Matched a")
case b:
    fmt.Println("Matched b")
}

// different types not allowed:
```

```go
// "Compile time error: cannot convert "2" (untyped string constant)
    to type int"
/*switch value {
case 1:
    fmt.Println("Matched 1")
case "2":
    fmt.Println("Matched string 2")
}*/

// type switch
var i2 interface{} = 4
switch i2.(type) {
case int:
    fmt.Println("Matched int")
case string:
    fmt.Println("Matched string")
case bool:
    fmt.Println("Matched bool")
case nil:
    fmt.Println("Matched nil")
}
```

Printed Output:

```
4 - How are case values specified?
Matched 3
Matched 3
Matched 1, 2, or 3
Matched b
Matched int
```

- **What is done about unrepresented expression values?**
  In Go, if none of the `case` values are matched, `default` clause might be used to catch these unrepresented expression values.

```go
fmt.Println("\n\n5 - What is done about unrepresented expression
    values?")
value = 100
switch value {
case 10:
    fmt.Println("Matched 10")
default:
    fmt.Println("No match for the given value")
}
```

Printed Output:

```
5 - What is done about unrepresented expression values?
No match for the given value
```

1.3 JavaScript

There are several ways to make a multiple-way selection in Javascript, the first and the one we will use in this assignment is with `switch-case`. Others are `dictionaries as object maps` and `if-elif-else chains`. The one we will use `switch-case` in this assignment.

- **What is the form and type of the control expression?**
  In JavaScript, the control expression of a switch statement can be a variable, a literal, an expression. JavaScript uses strict equality (===) for comparing the control expression with case values, meaning types must match exactly. Object comparisons only succeed if they reference the exact same object in memory, which makes it impractical. The type-switch can also be implicitly achieved by finding the types of elements using the typeof operator.

```
class Obj {
    constructor(x) {this.x = x;}
}

console.log("1-What is the form and type of the control expression?")
    ;

// variable
let value = 5;
switch (value) {
    case 1:
    console.log("Control expression is a number with value 1");
    break;
    case 5:
    console.log("Control expression is a number with value 5");
    break;
}

// expression
let val1 = 3;
let val2 = 4;
switch (val1 + val2) {
    case 7:
    console.log("Control expression is an expression resulting in 7")
        ;
    break;
}

// literal
switch (10) {
    case 10:
    console.log("Control expression is a literal number 10");
    break;
    case 20:
    console.log("Control expression is a literal number 20");
    break;
}

// string
let str = "hello";
switch (str) {
    case "hello":
    console.log("Control expression is a string: hello");
    break;
    case "world":
    console.log("Control expression is a string: world");
    break;
}

// boolean
let isTrue = true;
switch (isTrue) {
    case true:
    console.log("Control expression is a boolean: true");
    break;
    case false:
    console.log("Control expression is a boolean: false");
    break;
}

// object
let obj = new Obj(4);
switch (obj) {
    case new Obj(4):
    console.log("This won't match because objects are compared by
       reference");
    break;
    default:
    console.log("Default case: Objects with same values but different
        references don't match");
    break;
}

// object property
switch (obj.x) {
    case 4:
    console.log("Control expression using object property works: obj.
       x = 4");
    break;
}

// null
let nullValue = null;
switch (nullValue) {
    case null:
    console.log("Control expression is null");
    break;
    default:
    console.log("Not null");
    break;
}

// Using undefined
let undefinedValue;
switch (undefinedValue) {
    case undefined:
    console.log("Control expression is undefined");
    break;
    default:
    console.log("Not undefined");
    break;
}

// typeof
let func = function() {};
let arr = [1, "two", true, null, [1, 2, 3], { a: 1, b: 2 }, func];
for (let i = 0; i < arr.length; i++) {
    let element = arr[i];
    console.log(`Checking element ${i}:`, element);

    switch (typeof element) {
        case "object":
            console.log("Control expression is an object (could be
               array, null, or plain object)");
            break;
        case "string":
            console.log("Control expression is a string");
            break;
```

```
          case "number":
              console.log("Control expression is a number");
              break;
          case "boolean":
              console.log("Control expression is a boolean");
              break;
          case "function":
              console.log("Control expression is a function");
              break;
          case "undefined":
              console.log("Control expression is undefined");
              break;
          case "symbol":
              console.log("Control expression is a symbol");
              break;
          case "bigint":
              console.log("Control expression is a bigint");
              break;
          default:
              console.log("Control expression is of an unknown type");
              break;
      }
}

let num = 7;
switch (true) {
  case num < 5:
    console.log("Less than 5");
    break;
  case num < 10:
    console.log("Less than 10");
    break;
  default:
    console.log("10 or more");
}
```

Printed Output:

```
1-What is the form and type of the control expression?
Control expression is a number with value 5
Control expression is an expression resulting in 7
Control expression is a literal number 10
Control expression is a string: hello
Control expression is a boolean: true
```

```
Default case: Objects with same values but different references don't match
Control expression using object property works: obj.x = 4
Control expression is null
Control expression is undefined
Checking element 0: 1
Control expression is a number
Checking element 1: two
Control expression is a string
Checking element 2: true
Control expression is a boolean
Checking element 3: null
Control expression is an object (could be array, null, or plain object)
Checking element 4: [ 1, 2, 3 ]
Control expression is an object (could be array, null, or plain object)
Checking element 5: { a: 1, b: 2 }
Control expression is an object (could be array, null, or plain object)
Checking element 6: [Function: func]
Control expression is a function
Less than 10
```

- **How are the selectable segments specified?**
  In JavaScript, selectable segments are defined using the case keyword, followed by a constant value and a colon :, then one or more actions. Actions can be single or multiple statements. If multiple cases share the same actions, they can be stacked without code between them. Optionally, curly braces can be used to group statements inside a case.

```
console.log("\n\n2-How are the selectable segments specified?")
value = 5;
switch (value) {
    case 5:
    console.log("single statement");
    break;
    case 10:
    console.log("first statement");
    console.log("multiple statment");
    break;
    case 15:
    case 20:
    console.log("multiple case statement");
    case 25: {
        console.log("case with block");
    }
}
```

Printed Output:

```
2-How are the selectable segments specified?
single statement
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  No. In JavaScript, there is fall-through execution, meaning that after a matching case is found, JavaScript continues executing all following cases until a break is encountered or the end of the switch is reached.

```
console.log("\n\n3-Is execution flow through the structure restricted
    to include just a single selectable segment?");

value = 1;
switch (value) {
    case 1:
    console.log("Value is 1");
    break;
    default:
    console.log("Default case");
}
switch (value) {
    case 1:
    console.log("Value is 1");
    default:
    console.log("Default case also executes");
}
```

Printed Output:

```
3-Is execution flow through the structure restricted to include just a single selectable segment?
Value is 1
Value is 1
Default case also executes
```

- **How are case values specified?**
  In JavaScript, case values can be literals (numbers, strings etc.) expressions, or variables whose values are evaluated during execution. Also it is allowed that using mixed types as case values, in same switch-case statement and type matching is strict in these. Also multiple case values can be used with stacking them.

```
console.log("\n\n4-How are case values specified?");

value = 3;
// Literals
console.log("Literal case values:");
switch (value) {
    case 1:
    console.log("Matched 1");
    case 2:
    console.log("Matched 2");
}

// expressions
switch (value) {
    case 1 + 1:
    console.log("Matched 2");
    break;
    case 2 + 1:
    console.log("Matched 3");
}

// var ables
let x = 2, y = 3;
switch (value) {
    case x:
    console.log("Matched x (2)");
    break;
    case x + y:
    console.log("Matched x + y (5)");
    break;
}

// String case values
let footballer = "Barella";
switch (footballer) {
    case "Barella":
    console.log("Matched Barella");
    break;
    case "Lautaro":
    console.log("Matched Lautaro");
}

// mixed
let mixedValue = "42";
switch (mixedValue) {
    case 42:
    console.log("Matched number 42");
    break;
    case "42":
```

```
    console.log("Matched string '42'");
    break;
    case true:
    console.log("Matched boolean true");
    break;
}

let team = "Inter";
switch (team) {
    case "Inter":
    case "Milan":
    console.log("Italian team");
    break;
    case "Barcelona":
    console.log("Spanish team");
    break;
}
```

Printed Output:

```
4-How are case values specified?
Literal case values:
Matched 3
Matched Barella
Matched string '42'
Italian team
```

- **What is done about unrepresented expression values?**
  In JavaScript, unrepresented expressions can be handled in default case, but this isn't mandotary.

```
console.log("\n\n5-What is done about unrepresented expression values
    ?");
value = 100;
switch (value) {
    case 10:
        console.log("Matched 10");
        break;
    default:
        console.log("Default case: No match for the given value");
}
```

Printed Output:

```
5-What is done about unrepresented expression values?
Default case: No match for the given value
```

1.4 Kotlin

In Kotlin, multiple-way selection is done with the `when` expression, which is the equivalent of switch in most languages. `when` can be used both as a statement and as an expression that returns a value and we will investigate this in this report. Kotlin also supports if-else if chains, which is a different way of doing multiple-way selection.

- **What is the form and type of the control expression?**
  In Kotlin, when expression supports literals, variables, expressions, objects etc. Nullable types might also be used as control expressions. Kotlin also supports direct object comparison based on structural equality. The when expression can even be used without an argument. In this case the cases become boolean expressions and the first case that is true is executed. Kotlin also provides sealed classes and enums, which provides smart casting capabilities that maintain type safety while eliminating type checking.

```
// variable
var value = 5
when (value) {
   1 -> println("Control expression is an integer with value 1")
   5 -> println("Control expression is an integer with value 5")
   else -> println("No match")
}

//expression
var val1 = 3
var val2 = 4
when (val1 + val2) {
   7 -> println("Control expression is an expression resulting in 7")
   else -> println("No match")
}

// literal
when (10) {
   10 -> println("Control expression is a literal with value 10")
   20 -> println("Control expression is a literal with value 20")
   else -> println("No match")
}

// object
var obj = Obj(4)
when (obj) {
   Obj(3) -> println("Object with x = 3")
   Obj(4) -> println("Object with x = 4")
   else -> println("No match")
}
```

```
// conditionals
when {
    value == 5 -> println("Value is 5")
    value > 10 -> println("Value is greater than 10")
    else -> println("Value is not 5 and not greater than 10")
}

// nullable types
val nullableValue: Int? = null
when (nullableValue) {
     null -> println("Value is null")
     1 -> println("Value is 1")
     else -> println("Value is neither null nor 1")
}

// enum
val colour = Color.RED
when (colour) {
     Color.RED -> println("Color is red")
     Color.BLUE -> println("Color is blue")
     else -> println("Color is not red or blue")
}

val items = listOf( Item.IntItem(1), Item.StringItem("apple"), Item.
    ArrayItem(arrayOf(3, 4)), Item.ListItem(listOf(1, 2, 3)), Item.
    MapItem(mapOf("key" to 1)), Item.BoolItem(true), Item.NothingItem,
     Item.ObjItem(Obj(3)))

for (item in items) {
    print("Value: $item, Type: ")
    when (item) {
        is Item.IntItem -> println("Control expression is an integer:
            ${item.value}")
        is Item.StringItem -> println("Control expression is a string:
             ${item.value}")
        is Item.ArrayItem -> println("Control expression is an array:
            ${item.value.contentToString()}")
        is Item.ListItem -> println("Control expression is a list: ${
            item.value}")
        is Item.MapItem -> println("Control expression is a map: ${
            item.value}")
        is Item.BoolItem -> println("Control expression is a boolean:
            ${item.value}")
        is Item.NothingItem -> println("Control expression is Nothing"
            )
        is Item.ObjItem -> {
            if (item.value.x == 3) {
                println("Control expression is an Obj with x = 3")
            } else {
                println("Control expression is an Obj with x = ${item.
                    value.x}")
            }
        }
    }
}
```

Printed Output:

```
1-What is the form and type of the control expression?
Control expression is an integer with value 5
Control expression is an expression resulting in 7
Control expression is a literal with value 10
Object with x = 4
Value is 5
Value is null
Color is red
Value: IntItem(value=1), Type: Control expression is an integer: 1
Value: StringItem(value=apple), Type: Control expression is a string: apple
Value: ArrayItem(value=[3, 4]), Type: Control expression is an array: [3, 4]
Value: ListItem(value=[1, 2, 3]), Type: Control expression is a list: [1, 2, 3]
Value: MapItem(value={key=1}), Type: Control expression is a map: {key=1}
Value: BoolItem(value=true), Type: Control expression is a boolean: true
Value: Item$NothingItem@300ffa5d, Type: Control expression is Nothing
Value: ObjItem(value=Obj(x=3)), Type: Control expression is an Obj with x = 3
```

- **How are the selectable segments specified?**
  In Kotlin, selectable segments defined as a value or condition followed by the **-** symbol, followed by after actions. Actions do not have to be enclosed in curly braces if they are single statements, but they can be. Actions must be enclosed in curly braces if they consist of multiple statements.

```
println("\n\n2-How are the selectable segments specified?")
when (value) {
    5 -> {println("Case 5: Single statement")}
    10 -> {
        println("Case 10: First statement")
        println("Case 10: Second statement - multiple statements in a
            block")
    }
    20 ->
        println("Case 20: First statement")
        //println("Case 20: Second statement - multiple statements in
            a block")
    else -> println("Default case")
}
```

Printed Output:

```
2-How are the selectable segments specified?
Case 5: Single statement
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  Yes, in Kotlin the first equal `case` blog is executed even if case is true, other cases below it are not executed, so there is no fall through execution. One value can be used in different cases.

```
println("\n\n3-Is execution flow through the structure restricted to
    include just a single selectable segment?")
value = 6
when (value) {
    2, 4, 6 -> println("Matched even number")
    3, 6, 9 -> println("Matched divisible by 3")
    else -> println("No match")
}
```

Printed Output:

```
3-Is execution flow through the structure restricted to include just a
    single selectable segment?
Matched even number
```

- **How are case values specified?**
  For case values, Kotlin allows different types like integers, strings etc. You can also use variables as case values. Also, Kotlin supports multiple case values, they can be in same case with comma separated. Kotlin also supports range patterns for matching continuous sequences of values, from x to y (inclusive).
  However, mixing different types of values in same `match-case` is not allowed in Kotlin since Kotlin has strong type checking. So types cannot mixed in when branches unless using `Any` type while determining the type of the value. This is called as smart casting, and since type of value is checked, you can now use it's classes functionalities.
  If a when expression does not take a specific arguement, then each branch condition can be a boolean expression, including function calls.

```kotlin
println("\n\n4-How are case values specified?")

val caseValue = 3
// Literal
when (caseValue) {
   1 -> println("Matched 1")
   2 -> println("Matched 2")
   3 -> println("Matched 3")
   else -> println("No match")
}

// Multiple values
when (caseValue) {
   1, 2, 3 -> println("Matched 1, 2, or 3")
   4, 5, 6 -> println("Matched 4, 5, or 6")
   else -> println("No match")
}

// Expressions
when (caseValue) {
   2 + 1 -> println("Matched 3 (2+1)")
   10 - 3 -> println("Matched 7 (10-3)")
   else -> println("No match")
}

// Variables
val a = 1
val b = 3
when (caseValue) {
   a -> println("Matched a ($a)")
   b -> println("Matched b ($b)")
   else -> println("No match")
}

// different types not allowed
/*when (caseValue) {
    3 -> println("Matched integer 3")
    "3" -> println("This would cause a compilation error - cannot
       compare Int with String")
    else -> println("No match")
}*/
```

```
// Ranges
when (caseValue) {
   in 1..3 -> println("Matched range 1..3")
   !in 5..8 -> println("Matched not in range 5..8")
   else -> println("No match")
}

// conditionals
when {
    caseValue < 0 -> println("Negative value")
    caseValue == 0 -> println("Zero value")
    caseValue > 0 && caseValue <= 3 -> println("Positive value
       between 1 and 3")
    caseValue > 3 -> println("Positive value greater than 3")
}

// functions
fun isEven(n: Int) = n % 2 == 0
fun isOdd(n: Int) = n % 2 == 1
when {
    isEven(caseValue) -> println("$caseValue is even")
    isOdd(caseValue) -> println("$caseValue is odd")
}

// Type checking
val anyValue: Any = "Hello"
when (anyValue) {
   is String -> println("anyValue is a String: $anyValue, and its
      length is ${anyValue.length}")
   is Int -> println("anyValue is an Int: $anyValue")
   is Boolean -> println("anyValue is a Boolean: $anyValue")
   else -> println("anyValue is of some other type")
}
```

Printed Output:

```
4-How are case values specified?
Matched 3
Matched 1, 2, or 3
Matched 3 (2+1)
Matched b (3)
Matched range 1..3
Positive value between 1 and 3
3 is odd
```

```
anyValue is a String: Hello
```

- **What is done about unrepresented expression values?**
  In Kotlin, when used as an expression, `when` must be exhaustive — meaning it must handle all possible input values. If not all cases are covered, an else branch is required. When used as a statement (not returning a value), the else is optional.

```
println("\n\n5-What is done about unrepresented expression values?")
// else is optional
val unrepValue = 100
when (unrepValue) {
   10 -> println("Matched 10")
   else -> println("unrepresented value: $unrepValue")
}
when (unrepValue) {
    10 -> println("Matched 10")
}

// Expression requires an else branch or must be exhaustive
val resultWithElse = when (unrepValue) {
   10 -> "ten"
   20 -> "twenty"
   else -> "no match" // Required
}
println("Result with else: $resultWithElse")

// have to cover all cases
val color = Color.BLUE
val colorName = when (color) {
   Color.RED -> "Red"
   else -> "others"
}
println("Color name: $colorName")
```

Printed Output:

```
5-What is done about unrepresented expression values?
unrepresented value: 100
Result with else: no match
Color name: others
```

1.5 PHP

There are several ways to provide multiple-way selection in PHP: You can use traditional `switch-case` which evaluates an expression and compares the result against multiple case values. Also, there are `if-elseif-else chains`, `match expressions` which come to the language with PHP 8.0 and returns a value based on the matching case or lastly `array lookups` for simple value mapping. In this report `switch-case` is used since it is more similar to multiple-way selection statements in other languages. PHP doesn't have built-in type pattern matching, instead the switch statement evaluates the control expression once and compares it against the case values using loose comparison (==).

- **What is the form and type of the control expression?**
  In PHP, control expressions for match statements can be of any type, including native types (integers, booleans, strings etc.). PHP doesn't support directly using objects as control expressions in switch statements, but you can work with object properties instead. PHP doesn't directly support type switch, but PHP provides the gettype() function which returns a type as a string. Thus, PHP supports for a type-based switching mechanism undirectly. PHP also supports using expressions as control values.

```php
class Obj {
    public $x;
    public function __construct($x) {
        $this->x = $x;
    }
}

echo "1-What is the form and type of the control expression?";

// variable
$value = 5;
switch ($value) {
    case 1:
        echo "Control expression is an integer\n";
        break;
    case 5:
        echo "Control expression is an integer with value 5\n";
        break;
}

// expression
$val1 = 3;
$val2 = 4;
switch ($val1 + $val2) {
    case 7:
        echo "Control expression is an expression resulting in 7\n";
        break;
}
```

```
// literal
switch (10) {
    case 10:
        echo "Control expression is a literal\n";
        break;
    case 20:
        echo "Control expression is a literal with value 20\n";
        break;
}

// string
switch ("hello") {
    case "hi":
        echo "Control expression is a string 'hi'\n";
        break;
    case "hello":
        echo "Control expression is a string 'hello'\n";
        break;
}


// object not supported directly
$obj = new Obj(4);
switch ($obj->x) {
    case 3:
        echo "Object has x = 3\n";
        break;
    case 4:
        echo "Object has x = 4\n";
        break;
}

// PHP doesn't have built-in type switching, but we can use gettype()
    for similar behavior
$items = [1, "apple", [3, 4], [1, 2, 3], ["key" => 1], true, NULL,
   new Obj(3)];

foreach ($items as $item) {
    $type = gettype($item);
    echo "Value: ";
    if ($type == "array") {
        echo json_encode($item);
    } elseif ($type == "object") {
        echo "Object";
    } elseif ($type == "NULL") {
        echo "NULL";
    } else {
        echo $item;
    }
    echo ", Type: $type\t\t";

    switch ($type) {
        case "integer":
            echo "Control expression is an integer\n";
            break;
        case "string":
            echo "Control expression is a string\n";
            break;
        case "array":
            if (count($item) == 2 && isset($item[0]) && isset($item
               [1]) && !isset($item[2])) {
                echo "Control expression is an array length 2\n";
            } elseif (isset($item["key"])) {
                echo "Control expression is an associative array\n";
            } else {
                echo "Control expression is a sequential array\n";
            }
            break;
        case "boolean":
            echo "Control expression is a boolean\n";
            break;
        case "NULL":
            echo "Control expression is NULL\n";
            break;
        case "object":
            if ($item instanceof Obj && $item->x == 3) {
                echo "Control expression is an Obj with x = 3\n";
            } else {
                echo "Control expression is an object\n";
            }
            break;
        default:
            echo "No match found\n";
    }
}
```

Printed Output:

```
1-What is the form and type of the control expression?Control expression is an integer with value 5
Control expression is an expression resulting in 7
Control expression is a literal
Control expression is a string 'hello'
Object has x = 4
Value: 1, Type: integer  Control expression is an integer
Value: apple, Type: string  Control expression is a string
Value: [3,4], Type: array  Control expression is an array length 2
Value: [1,2,3], Type: array  Control expression is a sequential array
Value: {"key":1}, Type: array  Control expression is an associative array
Value: 1, Type: boolean  Control expression is a boolean
Value: NULL, Type: NULL  Control expression is NULL
Value: Object, Type: object  Control expression is an Obj with x = 3
```

- **How are the selectable segments specified?**
  In PHP, selectable segments are specified as case blocks. Each case block starts with the case keyword followed by a value and a colon and ends at the next case or at the end of a switch block. Statements within a case block can contain more than one statement, and you can have empty case blocks that move to the next case. PHP also allows empty cases, which can be useful for handling multiple values with the same code block.

```php
echo "\n\n2-How are the selectable segments specified?\n";
$value = 10;
switch ($value) {
    case 5:
        echo "Case 5: Single statement\n";
        break;
    case 10:
        echo "Case 10: First statement\n";
        echo "Case 10: You can have multiple statements in a case
           block\n";
        break;
}

// Empty case
$value = 3;
switch ($value) {
    case 3:
        // fall through
    case 4:
        echo "This is case 3 or 4.\n";
        break;
    case 5:
        echo "This is case 5 with statement\n";
        break;
}
```

Printed Output:

```
2-How are the selectable segments specified?
Case 10: First statement
Case 10: You can have multiple statements in a case block
This is case 3 or 4.
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  No. PHP follows a "fall-through" behavior where execution continues to subsequent cases unless explicitly stopped with a break statement.

```
echo "\n\n3-Is execution flow through the structure restricted to
   include just a single selectable segment?\n";

$value = 6;
switch ($value) {
    case 2:
    case 4:
    case 6:
        echo "Matched even number\n";
    case 3:
    case 6:
    case 9:
        echo "Matched current case or fell through from even number
           case\n";
    default:
        echo "No match\n";
}

$value = 6;
switch ($value) {
    case 2:
    case 4:
    case 6:
        echo "Matched even number\n";
        break;
    case 3:
    case 6:
    case 9:
        echo "Matched divisible by 3\n";
        break;
    default:
        echo "No match\n";
}
```

Printed Output:

```
3-Is execution flow through the structure restricted to include just a
    single selectable segment?
Matched even number
Matched current case or fell through from even number case
No match
Matched even number
```

- **How are case values specified?**
  For case values, PHP allows different types like integers, strings etc. You can also use variables as case values. PHP also supports type cohersion, so it tries to convert types to fit value. Also cases might be conditions or functions.

```
echo "\n\n4-How are case values specified?\n";
$value = 3;
// Literals
switch ($value) {
    case 2:
        echo "Matched 2\n";
        break;
    case 3:
        echo "Matched 3\n";
        break;
}

// Expressions
switch ($value) {
    case 2 + 1:
        echo "Matched 3 (2+1)\n";
        break;
    case 10 - 3:
        echo "Matched 7 (10-3)\n";
        break;
}

// Variables
$m = 2; $n = 3;
switch ($value) {
    case $m:
        echo "Matched m ($m)\n";
        break;
    case $m * $n:
        echo "Matched m * n ($m * $n)\n";
        break;
}

$value = "3";
switch ($value) {
    case 3:
        echo "Matched integer 3 with string '3' due to type coercion\
           n";
        break;
    case "3":
        echo "Matched string '3'\n";
        break;
}

// conditionals
$value = 6;
function isEven($num) {return $num % 2 == 0;}
switch ($value) {
    case isEven($value):
        echo "Matched condition (isEven(value))\n";
        break;
    case ($value > 4):
        echo "Matched condition (value > 4)\n";
        break;
    default:
        echo "No match\n";
}
```

Printed Output:

```
4-How are case values specified?
Matched 3
Matched 3 (2+1)
Matched integer 3 with string '3' due to type coercion
Matched condition (isEven(value))
```

- **What is done about unrepresented expression values?**
  For unrepresented values, PHP supports case `default`, if a value is not matched with any case until it encounter with `default` case, it enters the default's block.

```
echo "\n\n5-What is done about unrepresented expression values?\n";
```

```php
$value = 100;
// without default
switch ($value) {
    case 10:
        echo "Matched 10\n";
}

// With default
switch ($value) {
    case 10:
        echo "Matched 10\n";
    default:
        echo "No matching, default case executed\n";
}
```

Printed Output:

```
5-What is done about unrepresented expression values?
No matching, default case executed
```

1.6 Python

There are several ways to make a multiple-way selection in Python, the first and the one we will use in this assignment is with `match-case`. Others are `dictionaries as function maps` and `if-elif-else chains`. The one we will use `match-case`, comes with Python 3.10.

- **What is the form and type of the control expression?**
  In Python, multiple-way selections are made with `match-case`, the control expression used in a `match` statement can be any valid Python expression, including integers, strings, tuples, lists, dictionaries, enums, and even custom objects. The expression we write in `match` is compared with all `cases` from top to bottom, and the first matching `case` is found (if exists) and executed.

```python
print("1-What is the form and type of the control expression?")

# variable
value = 5
match value:
    case 1:
        print("Control expression is an integer")
    case "apple":
        print("Control expression is a string")

# expression
val1 = 3
val2 = 4
match val1 + val2:
    case 7:
        print("Control expression is an integer")
    case "apple":
        print("Control expression is a string")


# data structure
match (val1, val2):
    case (1, 2):
        print("Control expression is a tuple")
    case [1, 2]:
        print("Control expression is a list")

# enum
class fruit(Enum):
    apple = 1
    banana = 2
value = fruit.apple
```

```
match value:
    case fruit.apple:
        print("Control expression is an enum")

class obj:
    def __init__(self,x):
        self.x = x
l = [1, "apple", (3, 4), [1, 2, 3], {"key": "value"}, range(1, 10),
   {1, 2, 3}, True, None, obj(3)]

for val in l:
    print(f"\nTesting value: {val}, type: {type(val)}")
    match val:
        case 1:
            print("Control expression is an integer")
        case "apple":
            print("Control expression is a string")
        case (3, 4):
            print("Control expression is a tuple")
        case [1, 2, 3]:
            print("Control expression is a list")
        case {"key": "value"}:
            print("Control expression is a dictionary")
        case range():
            print("Control expression is a range")
        case True:
            print("Control expression is a boolean")
        case None:
            print("Control expression is None")
#         case obj(5):
#             print("Control expression is an object")
        case obj() as o if o.x == 3:
            print("Control expression is an object with x = 3")
        case _:
            print("No match found")
```

Printed Output:

```
1-What is the form and type of the control expression?
Control expression is an integer
Control expression is an enum

Testing value: 1, type: <class 'int'>
Control expression is an integer
```

```
Testing value: apple, type: <class 'str'>
Control expression is a string

Testing value: (3, 4), type: <class 'tuple'>
Control expression is a tuple

Testing value: [1, 2, 3], type: <class 'list'>
Control expression is a list

Testing value: {'key': 'value'}, type: <class 'dict'>
Control expression is a dictionary

Testing value: range(1, 10), type: <class 'range'>
Control expression is a range

Testing value: {1, 2, 3}, type: <class 'set'>
No match found

Testing value: True, type: <class 'bool'>
Control expression is an integer

Testing value: None, type: <class 'NoneType'>
Control expression is None

Testing value: <__main__.obj object at 0x0000016385B40D70>, type: <class '__main__.obj'>
Control expression is an object with x = 3
```

- **How are the selectable segments specified?**
  Each case must be followed by an indented statement block (as per Python's indentation rules). This statement block can contain one or more Python statement. It executes only if the case pattern is matched and Python control flow enters that block.

```python
print("\n\n2-How are the selectable segments specified?")
value = 5
match value:
    case 5:
        l.append("Python is love")
        print(l)
        print("Multiple statements in a single case")
    case 10:
        print("Single statement in a case")
```

Printed Output:

```
2-How are the selectable segments specified?
[1, 'apple', (3, 4), [1, 2, 3], {'key': 'value'}, range(1, 10), {1, 2, 3}, True, None,
    <__main__.obj object at 0x00000164976D6A50>, 'Python is love']
Multiple statements in a single case
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  Yes, in Python the first equal `case` blog is executed, other cases below it are not executed, so there is no fall through execution.

```
print("\n\n3-Is execution flow through the structure restricted to
    include just a single selectable segment?")
value = 6
match value:
    case 2 | 4 | 6:
        print("Matched even number")
    case 3 | 6 | 9:
        print("Matched divisible by 3")
    case _:
        print("No match")
```

Printed Output:

```
3-Is execution flow through the structure restricted to include just a single selectable
    segment?
Matched even number
```

- **How are case values specified?**
  In Python, selectable segments are specified with the values given after `case`. Each `case` represents a case where the value can be equal. If the case is true, the statements of this `case` are executed. Case values can be literals or multiple values can be combined using the OR operator (—). Python also supports pattern matching for data structures like tuples, lists when they used as case values. Also case values might be used with conditions for filtering more.

```
print("\n\n4-How are case values specified?")
value = "hello"
variable = "variable"
match value:
    case "hello": # string literal
        print("Matched 'hello'")
    case 32: # integer literal
        print("Matched Isparta")
    case "Mugla" | | 48: # different types in a single case
        print("Matched Mugla")
    case (34, 35): # in a data structure
        print("Matched tuple")
    case str() as s if s.isalpha():
        print(f"Matched string: {s}")
    case obj() as o if o.x == 3:
            print("Matched obj with x = 3")
    case _:
        print("No match")
```

Printed Output:

```
4-How are case values specified?
Matched 'hello'
```

- **What is done about unrepresented expression values?**
  If `value` does not match any case, a case (**case** _) can be written to capture the default values.

```
print("\n\n5-What is done about unrepresented expression values?")
value = 100
match value:
    case 10:
        print("Matched 10")
    case _:
        print("No match for the given value")
```

Printed Output:

```
5-What is done about unrepresented expression values?
No match for the given value
```

1.7 Rust

There are several ways to make a multiple-way selection in Rust, the first and the one we will use in this assignment is with `match-case`. It can be done with if-elif-else chains, if let, and while let. It is mainly focused on `match-case` in this report. `Match-case`'s supported features are wide, thus, for handling multiple types in same `match-case`, I will only use enum based approach. For future work, `any type with downcasting` or `generics` might be focused.

- **What is the form and type of the control expression?**

  In Rust, control expressions for match statements can be of any type, including native types (integers, booleans, strings etc.) as well as complex types like structs, enums, and reference types. The type system is strictly enforced, and patterns must be compatible with the control expression's type. For using different types in the same match statement, you can use enums in Rust since enums accept many types. You can also use built-in enum `Option`, which shows either the presence or the absence of a value. When you have an Option type, you can use pattern matching to handle both cases.

```
println!("1 - What is the form and type of the control expression?");

// variable
let value = 5;
match value {
    1 => println!("Control expression is an integer"),
    5 => println!("Control expression is an integer"),
    _ => (),
}

// expression
let val1 = 3;
let val2 = 4;
match val1 + val2 {
    7 => println!("Control expression is an expression"),
    _ => (),
}

// literal
match 10 {
    10 => println!("Control expression is a literal"),
    20 => println!("Control expression is a literal"),
    _ => (),
}

// object
#[derive(PartialEq, Debug)]
struct Obj {
    x: i32,
}
let obj = Obj { x: 4 };
match obj {
    Obj { x: 3 } => println!("Object with x = 3"),
    Obj { x: 4 } => println!("Object with x = 4"),
    _ => (),
}

// Option type
let some_value: Option<i32> = Some(5);
match some_value {
    Some(n) => println!("Got a value: {}", n),
    None => println!("Got None"),
}

// Match on different types using enums
println!("\nExample with different types using enums:");

#[derive(Debug)]
enum Item {
    Integer(i32),
    Text(String),
    Array([i32; 2]),
    Vector(Vec<i32>),
    Map(std::collections::HashMap<String, i32>),
    Boolean(bool),
    Nothing,
    CustomObj(Obj),
}

let items = vec![
    Item::Integer(1),
    Item::Text(String::from("apple")),
    Item::Array([3, 4]),
    Item::Vector(vec![1, 2, 3]),
    {
        let mut map = std::collections::HashMap::new();
        map.insert(String::from("key"), 1);
        Item::Map(map)
    },
    Item::Boolean(true),
    Item::Nothing,
    Item::CustomObj(Obj { x: 3 }),
];

for item in items {
    print!("Value: {:?}, Type: ", item);
    match item {
        Item::Integer(n) => println!("Control expression is an
            integer: {}", n),
        Item::Text(s) => println!("Control expression is a string: {}
            ", s),
        Item::Array(arr) => println!("Control expression is an array
            length 2: {:?}", arr),
        Item::Vector(vec) => println!("Control expression is a vector
            : {:?}", vec),
        Item::Map(map) => println!("Control expression is a map: {:?}
            ", map),
        Item::Boolean(b) => println!("Control expression is a boolean
            : {}", b),
        Item::Nothing => println!("Control expression is Nothing"),
        Item::CustomObj(obj) => {
            if obj.x == 3 {
                println!("Control expression is an Obj with x = 3")
            } else {
                println!("Control expression is an Obj")
            }
        }
    }
}
```

Printed Output:

```
1 - What is the form and type of the control expression?
Control expression is an integer
Control expression is an expression
Control expression is a literal
Object with x = 4
Got a value: 5

Example with different types using enums:
Value: Integer(1), Type: Control expression is an integer: 1
Value: Text("apple"), Type: Control expression is a string: apple
Value: Array([3, 4]), Type: Control expression is an array length 2: [3, 4]
Value: Vector([1, 2, 3]), Type: Control expression is a vector: [1, 2, 3]
Value: Map({"key": 1}), Type: Control expression is a map: {"key": 1}
```

```
Value: Boolean(true), Type: Control expression is a boolean: true
Value: Nothing, Type: Control expression is Nothing
Value: CustomObj(Obj { x: 3 }), Type: Control expression is an Obj with x = 3
```

- **How are the selectable segments specified?**
  In Rust, selectable segments are specified by patterns and they are followed by the =¿ operator, which connects the pattern to its corresponding action. Action parts can contain either a single statement or a block of multiple statements. All action parts might be in a curly braces, but if there is only a single statement, is is legal to use without curly braces, but it is still needed to add comma after the action. For the action part at least one expression is needed, it can't be empty but if no actions, you can just use ().

```rust
println!("\n\n2 - How are the selectable segments specified?");
let value = 5;
match value {
    5 => println!("Case 5: Single statement"),
    10 => {
        println!("Case 10: Multiple statements");
        println!("You can have more than one line in a case block");
    }
    20 =>{
        println!("Case 20: single statement");
    }
    _ => (),
}
```

  Printed Output:

```
2 - How are the selectable segments specified?
Case 5: Single statement
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  Yes, in Rust the first equal `case` blog is executed even if case is true, other cases below it are not executed, so there is no fall through execution.

```rust
println!("\n\n3 - Is execution flow restricted to just a single
    selectable segment?");
let value = 6;
match value {
    2 | 4 | 6 => println!("Matched even number"),
    3 | 6 | 9 => println!("Matched divisible by 3"),
    _ => println!("No match"),
}
```

  Printed Output:

```
3 - Is execution flow restricted to just a single selectable segment?
Matched even number
```

- **How are case values specified?**
  For case values, Rust allows different types like integers, strings etc. You can also use variables as case values. Also, Rust supports multiple case values, they can be in same case with comma separated. Rust also supports range patterns with the syntax x..=y for matching continuous sequences of values, from x to y (inclusive). For enums cases are the states of enums, and in cases you can also hold the value of enum in a variable. Also, `cases` allow runtime condition checking by adding an if clause inside them with syntax `x if condition`. However, mixing different types of values in same `match-case` is not allowed in Rust.

```
println!("\n\n4 - How are case values specified?");

let value = 3;
// literal values
match value {
    1 => println!("Matched 1"),
    2 => println!("Matched 2"),
    3 => println!("Matched 3"),
    _ => (),
}

// variables
let three = 2 + 1;
let seven = 10 - 3;
match value {
    three => println!("Matched 3"),
    seven => println!("Matched 7"),
    _ => (),
}

let strValue = "Hello";
match strValue {
    "Hello" => println!("Matched Hello"),
    "World" => println!("Matched World"),
    _ => (),
}

// different types
/*let value = 4;
match value {
    "Hello" => println!("Different types are not allowed"),
    4 => println!("Different types are not allowed"),
    4.0 => println!("Different types are not allowed"),
    _ => (),
}*/

// multiple case values using OR patterns
match value {
    1 | 2 | 3 => println!("Matched 1, 2, or 3"),
    4 | 5 | 6 => println!("Matched 4, 5, or 6"),
    _ => (),
}

// Pattern matching with ranges
for i in 1..10 {
    match i {
        1..=3 => println!("Matched range 1 to 3: {}", i),
        4..=6 => println!("Matched range 4 to 6: {}", i),
        _ => println!("No match for: {}", i),
    }
}

// Type matching with enums
#[derive(Debug)]
enum Value {
    Int(i32),
    Str(String),
    Bool(bool),
}

let val = Value::Int(4);
match val {
    Value::Int(i) => println!("Matched int: {}", i),
    Value::Str(s) => println!("Matched string: {}", s),
    Value::Bool(b) => println!("Matched bool: {}", b),
}

// expression with variables
let m: i32 = 2;
let n: i32 = 3;
match value {
    x if x == m * n => println!("Matched 6"),
    x if x == m + n => println!("Matched 5"),
    x => println!("Matched other value: {}", x),
}
```

```rust
// variables with binding
let a: i32 = 1;
let b: i32 = 3;
match value {
    x if x == a => println!("Matched a"),
    x if x == b => println!("Matched b"),
    _ => (),
}
```

Printed Output:

```
4 - How are case values specified?
Matched 3
Matched 3
Matched Hello
Matched 1, 2, or 3
Matched range 1 to 3: 1
Matched range 1 to 3: 2
Matched range 1 to 3: 3
Matched range 4 to 6: 4
Matched range 4 to 6: 5
Matched range 4 to 6: 6
No match for: 7
No match for: 8
No match for: 9
Matched int: 4
Matched other value: 3
Matched b
```

- **What is done about unrepresented expression values?**
  In Rust, unrepresented expressions have to be dealed as a case. The case can be named as _ if the value is not going to be used, or with a variable if value is going to be used. If the default case is not used, Rust gives a compile time error.

```rust
println!("\n\n5 - What is done about unrepresented expression values?
   ");
let value = 100;
match value {
    10 => println!("Matched 10"),
    _ => println!("No match for the given value (using wildcard)"),
}
match value {
    10 => println!("Matched 10"),
    default_value => println!("No match for value: {}", default_value
        ),
}
/*match value {
    10 => println!("Matched 10"), // not compiled
}*/
```

Printed Output:

```
5 - What is done about unrepresented expression values?
No match for the given value (using wildcard)
No match for value: 100
```

2 Evaluation

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

2.1 Dart

Dart's `switch-case` logic is similar to C-style languages, which makes the language easy to get used to. However, the `flow through` logic in Dart is very complex, sometimes the break meaningful and sometimes it doesn't, which greatly reduces the readability of the language. Also, the fact that you can't write multiple values in the same case reduces the writability of the language. In addition, Dart's strict requirement that case values must be compile-time constants sometimes limits flexibility and can be a little frustrating.

2.2 Go

Go's switch case logic is similar to other languages and this makes the language easy to get used to. A control expression can be any type of expression, which increases the writability of the language. Also, Go does not have fall-through, and if fall-through is wanted, it can be provided with keyword, which makes the language easier. The fact that it doesn't allow a case value to be in more than one place makes it less writeable but more readable. The presence of the type-switch also increases the flexibility of the language.

2.3 JavaScript

JavaScript's `switch-case` is very similar to C-style languages, and this makes language easy to use. Also, JavaScript allows to specify case values and control expression, which makes it easy to write and read. But, it allows all-through execution which makes it hard to read. Additionally, JavaScript compares values with strict equality (`===`) unlike Python, so there are no unexpected bugs.

2.4 Kotlin

Kotlin's `when` expression is one of the cleanest and most readable implementations of multiple-way selection. It works like a switch but extends it with the ability to match expressions, types, ranges, and conditions. Although it increases the reliability of the language that it does not allow more than one type in the same switch case (except in special cases), fact that the same value can be in more than one case decreases the reliability. Also, for expression type switch cases it requres an else branch or must be exhaustive, and that makes language more robust.

2.5 PHP

PHP's traditional `switch-case` is very familiar to developers coming from C/Java backgrounds, but its behavior introduces problems: loose type comparisons (using `==`) can lead to subtle, unexpected matches. Readability is moderate because while the syntax is simple, the loose typing introduces potential confusion. writability is easy due to PHP's forgiving nature, but at the cost of precision. Fall-through is allowed by default, which requires careful usage of `break` to avoid mistakes.

Although PHP, like other languages, supports many things as control expressions, the restrictions on case values reduce the writability of the language. Also it allows fall through, which reduces the readability of the language. In addition, the fact that case values cannot be used together in the same case makes the language difficult to write. Finally, the language loose type comparisons (using `==`) can lead to unexpected matches, which makes it hard to write, and not reliable.

2.6 Python

While you can use many types as control expressions in Python, the way the language checks their equivalence with case values is very different from convention. It can do binding etc. when checking, which makes it a very difficult language to write and read, and it can also create bugs in the code.

2.7 Rust

The match statement in Rust is also very similar to other languages, and there are many types supported in match expression, and this makes Rust very writeable. Rust also supports features like type-switch, which makes it easy to use. However, I think the fact that the same value can be used in multiple cases makes the language unpredictable and hard to read.

2.8 Conclusion

The multiple-way selection logic in all these languages is very similar, which makes them generally close to each other. However, when I evaluate the differences between them based on a balance of readability, simplicity, flexibility, and practical safety, the most useful one for me is **Go**.

Go avoids the common fall-through mistakes found in other languages unless explicitly requested with `fallthrough`. Also, the flexibility of switch and case expressions in Go makes the language easy to use. Go provides all the features of other languages, but also places a lot of emphasis on safety, therefore I find Go more user-friendly than other languages.

3 Learning Strategy

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 multiple-way selection statements 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 multiple-way selection features across languages.
- **GitHub**: [4] For checking example projects, code snippets, and previous `programing languages homeworks`.
- **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 and practiced writing full source files to test each multi-way selection features 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. I also ensured that my sample programs were complete and self-contained.

3.3 Personal Communication

I got help from Ece Kunduracıoğlu, the teaching assistant of the course, to clarify some questions during the assignment. I would like to thank her 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.