CS315 - Homework 2 Report

Emine İrem Esendemir

ID: 22202103

Department of Computer Engineering

Bilkent University

Ankara, Turkey

April 6, 2025

1 Part A - Operations in each language

1.1 Dart

In here, for demonstration, the below enum type and function is used.

```
int test() {
    return 2;
}
enum EnumExample {
    A,
    B,
    C,
    D
}
```

- **What is the form and type of the control expression?**
  In Dart, the control expression in a `switch` statement is the value placed in parentheses right after the `switch` keyword. This expression is evaluated once and compared against each `case` value. The control expression can various forms, such as a variable, a literal, a function call, or the result of an arithmetic expression. Dart supports various types for the control expression, including `int`, `String`, `double`, and `enum` values.

```
int test() {
return 2;
}

print("1. What is the form and type of the control expression?");
print("1.1 Forms of control expression:");

// variable
var x = 2;
switch (x) {
case 1:
    print("x is 1");
    break;
case "2":
    print("x is 2");
    break;
default:
    print("default case executed");
}

// arithmetic expression
var y = 3;
```

```
switch (x + y) {
case 4:
    print("x + y is 4");
    break;
case 5:
    print("x + y is 5");
    break;
default:
    print("default case executed");
}

// literal
switch ("hello") {
case "hi":
    print("The string is 'hi'");
    break;
case "hello":
    print("The string is 'hello'");
    break;
default:
    print("default case executed");
}

// function call
switch (test()) {
case 1:
    print("func returned 1");
    break;
case 2:
    print("func returned 2");
    break;
default:
    print("default case executed");
}

print("1.2 Types of control expression:");

// int
int intValue = 3;
switch (intValue) {
case 1:
    print("intValue is 1");
    break;
```

```
case 3:
    print("intValue is 3");
    break;
default:
    print("default case executed");
}

// string
String strValue = "kemal";
switch (strValue) {
case "mustafa":
    print("strValue is mustafa");
    break;
case "kemal":
    print("strValue is kemal");
    break;
default:
    print("default case executed");
}

// double
double doubleValue = 1.5;
// Dart 3.0+ is needed for this
switch (doubleValue) {
case 1.0:
    print("doubleValue is 1.0");
    break;
case 1.5:
    print("doubleValue is 1.5");
    break;
default:
    print("default case executed");
}

// Enum
EnumExample enumValue = EnumExample.B;
switch (enumValue) {
    case EnumExample.A:
    print("Enum value is A");
    break;
    case EnumExample.B:
    print("Enum value is B");
    break;
```

```
    case EnumExample.C:
    print("Enum value is C");
    break;
    case EnumExample.D:
    print("Enum value is D");
    break;
}
```

Printed Output:

```
1. What is the form and type of the control expression?
1.1 Forms of control expression:
default case executed
x + y is 5
The string is 'hello'
func returned 2
1.2 Types of control expression:
intValue is 3
strValue is kemal
doubleValue is 1.5
Enum value is B
```

- **How are the selectable segments specified?**
  In Dart, each selectable segment within a `switch` statement is specified using the `case` keyword, followed by a value and a colon. Within a case block, it is allowed to include multiple statements, including variable declarations, assignments, and function calls. Moreover, if the switch is used in an expression, it returns a value and the syntax of switch expression changes. In a `switch` expression each `case` uses an arrow (`=>`) to associate a matching pattern with a result value. Unlike normal `switch` statements, `switch` expressions must cover all possible input values, either through explicit cases or a catch-all _ case.

```
print("\n\n2. How are the selectable segments specified?");
var value = 1;
switch (value) {
  case 1:
    print("case 1");
    var localVar = "inside case 1";
    print("Local variable: $localVar");
    value += 10;
    print("Value after modification: $value");

  case 2:
    print("case 2");
    break;
```

```
  default:
    print("default case executed");
}

print("Switch expressions:");
x = 2;
y = switch(x) {
  1 => 2,
  2 => 3,
  _ => 4
};
print("x = $x, y = $y");
```

Printed Output:

```
2. How are the selectable segments specified?
case 1
Local variable: inside case 1
Value after modification: 11
x = 2, y = 3
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  In Dart, execution flow within a `switch` statement is usually restricted to a single selectable segment. If a `case` block contains executable statement, execution will not fall through to the next case. However, if a `case` block has no statements before the next case label, Dart allows fallthrough between them. If it is needed to continue to another case, the continue keyword can be used.

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

print("3.1 If no statement exist in the case block, execution will
    fall through to the next case:");
var testValue = 1;
switch (testValue) {
  case 1:
  case 2:
    print("testValue is either 1 or 2");
  case 3:
    print("testValue is 3");
  default:
    print("default case executed");
}

print("3.2 Using continue with label:");
```

```
testValue = 1;
switch (testValue) {
  case 1:
    print("testValue is 1");
    testValue = 2;
    continue case3;
  case2:
  case 2:
    print("continues to case 2");
    continue case3;
  case3:
  case 3:
    print("continues to case 3");
    break;
  case 4:
    print("not reached");
    break;
  default:
    print("default case executed");
}

print("3.3 If a statement exist between cases, it will not fall
   through:");
testValue = 1;
switch (testValue) {
  case 1:
    print("testValue is 1");
  case 2:
    print("testValue is 2");
  case 3:
    print("testValue is 3");
  default:
    print("default case executed");
}
```

Printed Output:

```
3. Is execution flow through the structure restricted to include just a single selectable segment?
3.1 If no statement exist in the case block, execution will fall through to the next case:
testValue is either 1 or 2
3.2 Using continue with label:
testValue is 1
continues to case 3
3.3 If a statement exist between cases, it will not fall through:
```

`testValue is 1`

- **How are case values specified?**
  In Dart, `case` values in a `switch` statement are specified using constant expressions such as literals (numbers, strings). Each case must be a compile-time constant; variables cannot be used directly as `case` labels unless they are marked as `const`. Dart also supports matching lists and extracting elements using spread like `[1, ...]` and `[..., 3]`.

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

// literal
var name = "kemal";
switch (name) {
  case "mustafa":
    print("name is mustafa");
    break;
  case "kemal":
    print("name is kemal");
    break;
  default:
    print("default case executed");
}

// variable
var varName = "mustafa";
var secondName = "kemal";
switch (name) {
  /*case varName:
    print("name is mustafa");
    // Error: Not a constant expression.
    break;*/
  case "kemal":
    print("name is kemal");
    break;
  default:
    print("default case executed");
}

// using constants
const firstName = "kemal";
switch (name) {
  case firstName:
    print("name matches firstName constant");
    break;
```

```
  default:
    print("default case executed");
}


// type patterns
Object obj = [1, 2, 3];

switch (obj) {
  case int():
    print("obj is an int");
    break;
  case String():
    print("obj is a String");
    break;
  case List():
    print("obj is a List");
    break;
  default:
    print("obj is something else");
}

// list
var list = [1, 2, 3];
switch (list) {
  case [1, 2, 3]:
    print("List contains exactly [1, 2, 3]");
    break;
  case [1, ...]:
    print("List starts with 1");
    break;
  case [..., 3]:
    print("List ends with 3");
    break;
  default:
    print("list is something else");
}
```

Printed Output:

```
4. How are case values specified?
name is kemal
name is kemal
name matches firstName constant (kemal)
```

```
obj is a List
List contains exactly [1, 2, 3]
```

- **What is done about unrepresented expression values?**
  In Dart, when the value of the control expression in a `switch` statement does not match any of the specified `case` labels, execution falls to the `default` block if it is provided. If no `default` is provided and the value is unrepresented, then the `switch` block is skipped without executing any code. However, when using `switch` with enums, Dart can enforce exhaustiveness: if not all possible enum values are covered, the compiler gives error.

```dart
print("\n\n5. What is done about unrepresented expression values?");

print("5.1 Using default case:");
var unrepresented = 4;
switch (unrepresented) {
case 1:
    print("number is 1");
    break;
case 2:
    print("number is 2");
    break;
case 3:
    print("number is 3");
    break;
default:
    print("default case executed");
}

print("5.2 No default case provided:");
var A = EnumExample.A;
switch (A) {
case EnumExample.A:
    print("Enum value is A");
    break;
case EnumExample.B:
    print("Enum value is B");
    break;
// If default is not provided,  Error: The type 'EnumExample' is not
   exhaustively matched by the switch cases since it doesn't match '
   EnumExample.C'.
default:
    print("default case executed");
}
```

```
// non-exhaustive switches
switch (unrepresented) {
case 1:
    print("number is 1");
    break;
case 2:
    print("number is 2");
    break;
case 3:
    print("number is 3");
    break;
}
```

Printed Output:

```
5. What is done about unrepresented expression values?
5.1 Using default case:
default case executed
5.2 No default case provided:
Enum value is A
```

1.2 Go

In here for demonstration, this codes are written at first.

```
func test() int {
    return 2
}
type TestObject struct {
    Name string
}
```

- **What is the form and type of the control expression?**
  In Go, the control expression in a switch statement appears right after the `switch` keyword and is evaluated once. This control expression can take many forms, including variables, literals, function calls, and arithmetic or logical expressions. It can also be omitted entirely, making a tagless switch, in which each case clause contains a boolean expression that is evaluated in sequence until one is true. The types supported by Go as control expressions are broad, including primitive types such as int, string, and bool, and comparable custom types like structs. In addition to value switches, Go also provides type switches using an `interface` value with the `.(type)` syntax, which branches based on the type of the value stored in the interface.

```
fmt.Println("1. What is the form and type of the control expression?"
    )
fmt.Println("1.1 Forms of control expression:")

// variable
x := 2
switch x {
case 1:
    fmt.Println("x is 1")
case 2:
    fmt.Println("x is 2")
default:
    fmt.Println("not found")
}

y := 3
switch x + y {
case 4:
    fmt.Println("x + y is 4")
case 5:
    fmt.Println("x + y is 5")
default:
    fmt.Println("not found")
}
```

```go
// Literals
switch "a" {
case "b":
    fmt.Println("The string is 'b'")
case "a":
    fmt.Println("The string is 'a'")
default:
    fmt.Println("not found")
}

// function call
switch test() {
case 1:
    fmt.Println("func returned 1")
case 2:
    fmt.Println("func returned 2")
default:
    fmt.Println("not found")
}

fmt.Println("1.2 Types of control expression:")
// integer type
intValue := 3
switch intValue {
case 1:
    fmt.Println("intValue is 1")
case 3:
    fmt.Println("intValue is 3")
}

// string type
strValue := "a"
switch strValue {
case "a":
    fmt.Println("strValue is 'a'")
case "b":
    fmt.Println("strValue is 'b'")
}

// boolean type
boolValue := true
switch boolValue {
```

```
case true:
    fmt.Println("it is true")
case false:
    fmt.Println("it is false")
}

person1 := TestObject{Name: "Emine"}
person2 := TestObject{Name: " rem "}

switch person1 {
case person2:
    fmt.Println("Matched person2")
case TestObject{Name: "Emine"}:
    fmt.Println("Matched Emine")
default:
    fmt.Println("not found")
}

// Tagless switch
x = 1
switch {
case x < 0:
    fmt.Println("x is negative")
case x == 0:
    fmt.Println("x is zero")
case x > 0:
    fmt.Println("x is positive")
}

// Type switch
var typeSwitch1 interface{} = "a"
switch typeSwitch1.(type) {
case int:
    fmt.Println("typeSwitch1 is int")
case string:
    fmt.Println("typeSwitch1 is string")
case bool:
    fmt.Println("typeSwitch1 is bool")
case TestObject:
    fmt.Println("typeSwitch1 is TestObject")
default:
    fmt.Println("typeSwitch1 is an unknown type")
}
```

Printed Output:

```
1. What is the form and type of the control expression?
1.1 Forms of control expression:
x is 2
x + y is 5
The string is 'a'
func returned 2
1.2 Types of control expression:
intValue is 3
strValue is 'a'
it is true
Matched Emine
x is positive
typeSwitch1 is string
```

- **How are the selectable segments specified?**
  In Go, each selectable segment in a switch statement is introduced using the `case` keyword, followed by one or more expressions, seperated by commas if there are multiple expressions, and a colon (:) to indicate the start of the case block. The statements to be executed if the case matches are placed in a standard Go code block that follows the colon. These segments can contain any valid statements, including variable declarations, function calls, and arithmetic operations. The execution of a case block continues until the end of that block is reached—either when another case or default label is encountered, or when the closing brace (}) of the switch statement is reached.

```go
fmt.Println("\n\n2. How are the selectable segments specified?")
value := 2
switch value {
case 1:
    fmt.Println("1 is executed")
    value *= 2
    fmt.Println("Value doubled:", value)
case 2, 3, 4:
    fmt.Println("2, 3 or 4 is executed")
    local := "inside case 1"
    fmt.Println("Local variable:", local)
    fmt.Println("Function result:", test())
    value += 10
    fmt.Println("Value after modification:", value)
default:
    fmt.Println("Default case")
}   fmt.Println("Default case executed")
```

Printed Output:

```
2. How are the selectable segments specified?
2, 3 or 4 is executed
Local variable: inside case 1
Function result: 2
Value after modification: 12
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  In Go, the default is to restrict to only the first matching case. Once a match is found, the associated statements are executed, and the switch statement exits automatically without evaluating any cases. This ensures that only a single selectable segment is executed by default. However, if continued execution into the next case block is desired, Go provides the `fallthrough` keyword. When fallthrough is used at the end of a case block, control passes directly to the next case, regardless of whether its condition matches the control expression.

```
fmt.Println("\n\n 3. Is execution flow through the structure
   restricted to include just a single selectable segment?")
fmt.Println("3.1 no fallthrough:")
testValue := 1
switch testValue {
case 1:
    fmt.Println("testValue is 1")
    testValue = 2
case 2:  // This won't execute even though testValue becomes 2
    fmt.Println("testValue is 2")
/*case 1:
    fmt.Println("Case 1 matched again")
Error: duplicate case 1 (constant of type int) in expression switch
*/
}
fmt.Println("new testValue:", testValue)

fmt.Println("\n3.2 Using fallthrough:")
testValue = 1
switch testValue {
case 1:
    fmt.Println("Case 1 exectuted")
    fallthrough
case 2:
    fmt.Println("Execution continues to case 2")
    fallthrough
case 3:
    fmt.Println("Execution continues to case 3")
case 4:
```

```
    fmt.Println("This won't execute")
}
fmt.Println("testValue after fallthrough switch:", testValue)
```

Printed Output:

```
3. Is execution flow through the structure restricted to include just a single selectable segment?
3.1 no fallthrough:
testValue is 1
new testValue: 2
3.2 Using fallthrough:
Case 1 exectuted
Execution continues to case 2
Execution continues to case 3
testValue after fallthrough switch: 1
```

- **How are case values specified?**
  The case values appear after the `case` keyword. They need to be same type as the control expression, otherwise it gives error. They can be literal, variable or an expression. They can take multiple values seperated by commas. It takes booleans if the switch case does not have a control expression. Also it can be based on the types in type switch and type's can be specified after.

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

// literal
name := "irem"
switch name {
case "emine":
    fmt.Println("name is emine")
case "irem":
    fmt.Println("name is irem")
}

/*wrongtype := "irem"
switch wrongtype {
case 1:
    fmt.Println("wrongtype is 1")
default:
    fmt.Println("wrongtype is not 1")
}
// Error: cannot convert 1 (untyped int constant) to type string
*/

// variable
trialCase := "irem"
```

```
switch name {
case trialCase:
    fmt.Println("trialCase is same as name")
case "irem":
    fmt.Println("name is irem")
}

// expression
num := 5
num1 := 3
num2 := 2
switch num {
case num1 + num2:
    fmt.Println("num is equal to num1 + num2")
case 10 - 2:
    fmt.Println("num is equal to 10 - 2")
}

// multiple values
fmt.Println("\nMultiple values in a single case:")
multipleValueNum := 1
multipleValueNum1 := 1
multipleValueNum2 := 2
multipleValueNum3 := 3
switch multipleValueNum {
case multipleValueNum1, multipleValueNum2:
    fmt.Println("the number is one of these two")
case multipleValueNum3:
    fmt.Println("the number is 3")
}

// tagless switch
taglessValue := 1
switch {
case taglessValue < 0:
    fmt.Println("taglessValue is negative")
case taglessValue == 0:
    fmt.Println("taglessValue is zero")
case taglessValue > 0:
    fmt.Println("taglessValue is positive")
}
// Type switch
var i interface{} = "Hello"
```

```
switch i.(type) {
case int:
    fmt.Println("i is of type int")
case string:
    fmt.Println("i is of type string")
case bool:
    fmt.Println("i is of type bool")
case TestObject:
    fmt.Println("i is of type TestObject")
default:
    fmt.Println("i is of an unknown type")
}
```

Printed Output:

```
4. How are the case values specified?
name is irem
trialCase is same as name
num is equal to num1 + num2
the number is one of these two
taglessValue is positive
i is of type string
```

- **What is done about unrepresented expression values?**
  If there is a case, specified as `default:`. The unrepresented values executed after that. If the default case is not given, then nothing is executed ans program continues execution after switch block.

```
fmt.Println("\n5. Handling unrepresented expression values:")
fmt.Println("5.1 Using default case:")
defaultInt := 3
switch defaultInt {
case 1:
    fmt.Println("defaultInt is 1")
case 2:
    fmt.Println("defaultInt is 2")
default:
    fmt.Println("defaultInt is not 1 or 2")
}

fmt.Println("5.2 No default case provided:")
switch defaultInt {
case 1:
    fmt.Println("defaultInt is 1")
case 2:
    fmt.Println("defaultInt is 2")
```

```
}
```

Printed Output:

```
5. Handling unrepresented expression values:
5.1 Using default case:
defaultInt is not 1 or 2
5.2 No default case provided:
```

1.3 JavaScript

To use by various questions, this method is defined:

```
function test() {
    return 2;
}
```

- **What is the form and type of the control expression?**
  In JavaScript, the control expression can take various forms, such as a variable, a literal, a function call, or the result of an arithmetic expression. Language allows a wide range of types for the control expression, including `number`, `string`, `boolean`, and `float`. However, JavaScript compares values and types strictly, so type mismatches will prevent a match. Additionally, complex types like objects are not suitable for direct use in `case` labels, since object comparison in JavaScript is based on reference equality. This makes `switch` suitable for primitive types but unreliable for object-based comparisons.

```
console.log("1.What is the form and type of the control expression?")
    ;
console.log("1.1 Forms of control expression:");

// variable
let x = 2;
switch (x) {
    case 1:
        console.log("x is 1");
        break;
    case 2:
        console.log("x is 2");
        break;
    default:
        console.log("default case executed");
}

// arithmetic expression
let y = 3;
switch (x + y) {
    case 4:
        console.log("x + y is 4");
        break;
    case 5:
        console.log("x + y is 5");
        break;
    default:
        console.log("default case executed");
}
```

```
// literal
switch ("hello") {
    case "hi":
        console.log("The string is 'hi'");
        break;
    case "hello":
        console.log("The string is 'hello'");
        break;
    default:
        console.log("default case executed");
}

// function call
switch (test()) {
    case 1:
        console.log("func returned 1");
        break;
    case 2:
        console.log("func returned 2");
        break;
    default:
        console.log("default case executed");
}

console.log("1.2 Types of control expression:");

// number
let intValue = 3;
switch (intValue) {
    case 1:
        console.log("intValue is 1");
        break;
    case 3:
        console.log("intValue is 3");
        break;
    default:
        console.log("default case executed");
}

// string
let strValue = "kemal";
switch (strValue) {
```

```
    case "mustafa":
        console.log("strValue is mustafa");
        break;
    case "kemal":
        console.log("strValue is kemal");
        break;
    default:
        console.log("default case executed");
}

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

// floating point
let floatValue = 1.5;
switch (floatValue) {
    case 1.0:
        console.log("floatValue is 1.0");
        break;
    case 1.5:
        console.log("floatValue is 1.5");
        break;
    default:
        console.log("floatValue is neither 1.0 nor 1.5");
}

// object (will not work properly with direct comparison)
let objValue = { name: "John" };
switch (objValue) {
    case { name: "John" }:  // This won't match
        console.log("objValue has name John");
        break;
    default:
        console.log("each object is unique");
}
```

Printed Output:

```
1. What is the form and type of the control expression?
1.1 Forms of control expression:
x is 2
x + y is 5
The string is 'hello'
func returned 2
1.2 Types of control expression:
intValue is 3
strValue is kemal
boolValue is true
floatValue is 1.5
each object is unique
```

- **How are the selectable segments specified?**
  In JavaScript, each selectable segment in a `switch` statement is defined using the `case` keyword, followed by a value and a colon. The code block that should run if the case matches is placed after the colon and continues until a `break` statement is encountered or to the end of the switch statement. These code blocks can include multiple lines and support all types of statements.

```
console.log("\n\n2. How are the selectable segments specified?");
let value = 1;
switch (value) {
    case 1:
        console.log("case 1");
        let localVar = "inside case 1";
        console.log("Local variable:", localVar);
        value += 10;
        console.log("Value after modification:", value);
        break;
    case 2:
        console.log("case 2");
        break;

    default:
        console.log("default case executed");
}
```

Printed Output:

```
2. How are the selectable segments specified?
case 1
Local variable: inside case 1
Value after modification: 11
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  In JavaScript, execution flow within a `switch` statement is not restricted to a single case by default. If a matching `case` is found and there is no `break` statement, execution will continue to the following case blocks, regardless of whether their values match the control expression or not. This behavior can be used on purpose when multiple cases should execute the same code, but it may also cause problems.

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

console.log("3.1 Using break:");
let testValue = 1;
switch (testValue) {
    case 1:
        console.log("testValue is 1");
        testValue = 2;
        break;
    case 2:
        console.log("testValue is 2");
        break;
    case 3:
        console.log("testValue is 3");
        break;
    default:
        console.log("default case executed");
}

console.log("3.2 Without using break: ");
testValue = 1;
switch (testValue) {
    case 1:
        console.log("testValue is 1");
        testValue = 2;
    case 2:
        console.log("continues to case 2");
    case 3:
        console.log("continues to case 3");
        break;
    case 4:
        console.log("not reached");
        break;
    default:
        console.log("default case executed");
}
```

```
// multiple cases sharing the same code block
let number = 1;
switch (number) {
    case 1:
    case 2:
        console.log("number is either 1 or 2");
        break;
    case 3:
        console.log("number is 3");
        break;
    default:
        console.log("default case executed");
}
```

Printed Output:

```
3. Is execution flow through the structure restricted to include just a single selectable segment?
3.1 Using break:
testValue is 1
3.2 Without using break:
testValue is 1
continues to case 2
continues to case 3
number is either 1 or 2
```

- **How are case values specified?**
  In JavaScript, `case` values in a `switch` statement can be specified using literals, variables, or expressions. Case values can include strings, numbers, booleans, or any other type that supports comparison with the control expression. JavaScript uses strict equality (`===`) when comparing the control expression to each `case` value, so both the value and type must match.

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

// literal
let nameVar = "kemal";
switch (nameVar) {
    case "mustafa":
        console.log("name is mustafa");
        break;
    case "kemal":
        console.log("name is kemal");
        break;
    default:
        console.log("default case executed");
```

```
}

// using expressions in case values
const a = "a";
const b = "b";
switch (nameVar) {
    case "cd":
        console.log("name is cd");
        break;
    case a + b:
        console.log("name matches a + b (ab)");
        break;
    default:
        console.log("default case executed");
}

// strict equality (===)
let stringNumber = "1";
switch (stringNumber) {
    case 1:
        console.log("This won't match because === is used");
        break;
    case "1":
        console.log("stringNumber is '1' (string)");
        break;
    default:
        console.log("default case executed");
}
```

Printed Output:

```
3. Is execution flow through the structure restricted to include just a single selectable segment?
3.1 Using break:
testValue is 1
3.2 Without using break:
testValue is 1
continues to case 2
continues to case 3
```

- **What is done about unrepresented expression values?**
  In JavaScript, if the value of the control expression in a `switch` statement does not match any of the defined `case` labels, and a `default` case is present, the code inside the `default` block will be executed. However, the use of a `default` case is optional. If it is omitted and no case matches the control expression, the entire `switch` block is skipped and nothing is executed.

```
console.log("\n\n5. What is done about unrepresented expression
   values?");

console.log("5.1 Using default case:");
let unrepresented = 4;
switch (unrepresented) {
    case 1:
        console.log("number is 1");
        break;
    case 2:
        console.log("number is 2");
        break;
    case 3:
        console.log("number is 3");
        break;
    default:
        console.log("default case executed");
}

console.log("5.2 No default case provided:");
switch (unrepresented) {
    case 1:
        console.log("number is 1");
        break;
    case 2:
        console.log("number is 2");
        break;
    case 3:
        console.log("number is 3");
        break;
}
```

Printed Output:

```
5. What is done about unrepresented expression values?
5.1 Using default case:
default case executed
5.2 No default case provided:
```

1.4 Kotlin

The below function and enum is declared to use in the questions:

```
fun test(): Int {
    return 2
}
enum class EnumExample {
    A, B, C
}
```

- **What is the form and type of the control expression?**
  In Kotlin, the control expression used in a `when` statement can take various forms. It may be a variable, a literal, the result of an arithmetic expression, or a function call. The evaluated result of this expression is then compared against the case conditions defined in each branch of the `when` block. It also can be used without an argument, in which case each branch becomes a boolean condition, acting similar to an `if-else-if` chain. As for types, the control expression can be of any type, including `Int`, `String`, `Boolean`, `Float`, `List` etc. Language also allows structural comparisons of collections such as lists.

```
println("1. What is the form and type of the control expression?")
println("1.1 Forms of control expression:")

// variable
val x = 2
when (x) {
    1 -> println("x is 1")
    2 -> println("x is 2")
    else -> println("default case executed")
}

// arithmetic expression
val y = 3
when (x + y) {
    4 -> println("x + y is 4")
    5 -> println("x + y is 5")
    else -> println("default case executed")
}

// literal
when ("hello") {
    "hi" -> println("The string is 'hi'")
    "hello" -> println("The string is 'hello'")
    else -> println("default case executed")
}
```

```
// function call
when (test()) {
    1 -> println("func returned 1")
    2 -> println("func returned 2")
    else -> println("default case executed")
}

println("1.2 Types of control expression:")

// int
val intValue = 3
when (intValue) {
    1 -> println("intValue is 1")
    3 -> println("intValue is 3")
    else -> println("default case executed")
}

// string
val strValue = "kemal"
when (strValue) {
    "mustafa" -> println("strValue is mustafa")
    "kemal" -> println("strValue is kemal")
    else -> println("default case executed")
}

// boolean
val boolValue = true
when (boolValue) {
    true -> println("boolValue is true")
    false -> println("boolValue is false")
}

// float
val floatValue = 1.5
when (floatValue) {
    1.0 -> println("floatValue is 1.0")
    1.5 -> println("floatValue is 1.5")
    else -> println("floatValue is neither 1.0 nor 1.5")
}

val listValue = listOf(1, 2, 3)
when (listValue) {
    listOf(1, 2) -> println("listValue is [1, 2]")
```

```
    listOf(1, 2, 3) -> println("listValue is [1, 2, 3]")
    else -> println("no match")
}

// no argument
val age = 85
when {
    age < 18 -> println("You are young")
    age in 18..65 -> println("You are an adult")
    age > 65 -> println("You are old")
    else -> println("Invalid age")
}


val enumValue = EnumExample.A
when (enumValue) {
    EnumExample.A -> println("Enum value is A")
    EnumExample.B -> println("Enum value is B")
    EnumExample.C -> println("Enum value is C")
}
```

Printed Output:

```
1. What is the form and type of the control expression?
1.1 Forms of control expression:
x is 2
x + y is 5
The string is 'hello'
func returned 2
1.2 Types of control expression:
intValue is 3
strValue is kemal
boolValue is true
floatValue is 1.5
listValue is [1, 2, 3]
You are old
Enum value is A
```

- **How are the selectable segments specified?**
  In Kotlin, selectable segments in a `when` expression are defined using individual branches, each consisting of a condition followed by an arrow (`->`) and a code block. The associated block is enclosed in curly braces. In branch, any statement can be included.

```
println("\n\n2. How are the selectable segments specified?")
```

```
var value = 1
when (value) {
    1 -> {
        println("case 1")
        val localVar = "inside case 1"
        println("Local variable: $localVar")
        value += 10
        println("Value after modification: $value")
    }
    2 -> {
        println("case 2")
    }
    else -> {
        println("default case executed")
    }
}
```

Printed Output:

```
2. How are the selectable segments specified?
case 1
Local variable: inside case 1
Value after modification: 11
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  In Kotlin, the execution flow within a `when` expression is limited to a single branch. Once a case matches the control expression, only the statements in that branch are executed, and no further branches are considered. If the same expression appears in multiple case, a warning is displayed.

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

var testValue = 1
when (testValue) {
    1 -> {
        println("testValue is 1")
        testValue = 2 //  won't execute to the 2 branch
    }
    2 -> {
        println("testValue is 2")
    }
    3 -> {
        println("testValue is 3")
    }
```

```
    else -> {
        println("default case executed")
    }
}
testValue = 1
when (testValue) {
    1 -> {
        println("testValue is 1")
    }
    1 -> { //  warning: duplicate branch condition in 'when'.
        println("testValue is 1 again")
    }
    else -> {
        println("default case executed")
    }
}
```

Printed Output:

```
3. Is execution flow through the structure restricted to include just a single selectable segment?
testValue is 1
testValue is 1
```

- **How are case values specified?**
  In Kotlin, case values in a `when` expression can be specified using literals, variables, expressions, ranges, or types. Variables and arithmetic operations, can also be used as case values. Multiple values can be combined in a single branch using commas, and value ranges can be matched using the `in` keyword along with range operators like `1..10`. Also, Kotlin supports to check the types using `when` statements with the `is` keyword, It allows the `when` expression to do smart casting. It helps to treat the value as the matched type within that branch. Also, as Kotlin has strong type-checking, different types of values cannot be used for the different branches of the same statement.

```
println("\n\n4. How case values are specified:");
// literal
val name = "kemal"
when ("kemal") {
    "mustafa" -> println("name is mustafa")
    "kemal" -> println("name is kemal")
    else -> println("default case executed")
}

// variable
when ("kemal") {
    "mustafa" -> println("name is mustafa")
    name -> println("name is kemal")
```

```
    else -> println("default case executed")
}

// expression
val num = 5
val num1 = 3
val num2 = 2
when (num) {
    num1 + num2 -> println("num is equal to num1 + num2")
    num1 * num2 -> println("num is equal to num1 * num2")
    else -> println("default case executed")
}

// multiple values in a single case
val number = 1
when (number) {
    1, 2 -> println("number is either 1 or 2")
    3 -> println("number is 3")
    else -> println("default case executed")
}

// using ranges
val rangeNumber = 7
when (rangeNumber) {
    in 1..5 -> println("rangeNumber is between 1 and 5")
    in 6..10 -> println("rangeNumber is between 6 and 10")
    else -> println("rangeNumber is outside these ranges")
}

// smart casting after type check
val anyValue2: Any = "mustafa kemal ataturk"
when (anyValue2) {
    is String -> println("String length: ${anyValue2.length}")
    is Int -> println("Integer value: ${anyValue2 + 1}")
    else -> println("Unknown type")
}

/*val intexample = 5
when (intexample) {
    "irem" -> println("intexample is irem")
    5 -> println("intexample is 5")
    else -> println("default case executed")
} error: incompatible types 'Int' and 'String'.*/
```

Printed Output:

```
4. How case values are specified:
name is kemal
name is kemal
num is equal to num1 + num2
number is either 1 or 2
rangeNumber is between 6 and 10
String length: 21
```

- **What is done about unrepresented expression values?**
  In Kotlin, if a `when` statement does not include a matching branch for the given control expression, and no `else` branch is provided, then the unmatched value is ignored and nothing is executed. However, if the `when` construct is used to return a value—then it must be exhaustive. This means all possible values must be covered, either explicitly or with an `else` branch. If the expression is not exhaustive, the compiler will give an error.

```
println("\n\n5. What is done about unrepresented expression values?")

println("5.1 Using else branch:")
val unrepresented = 4
when (unrepresented) {
    1 -> println("number is 1")
    2 -> println("number is 2")
    3 -> println("number is 3")
    else -> println("default case executed")
}

println("5.2 No else branch provided:")
val unrepresented2 = 4
when (unrepresented2) {
    1 -> println("number is 1")
    2 -> println("number is 2")
    3 -> println("number is 3")
}

// in an expression, must cover all.
val enumValue2 = EnumExample.B
val enumValuePrint = when (enumValue2) {
    EnumExample.A -> "AAAA"
    EnumExample.B -> "BBBB"
    EnumExample.C -> "CCCV"
}
//if not included all, gives error:  'when' expression must be
   exhaustive.
```

```
println("enumValuePrint: $enumValuePrint")
```

Printed Output:

```
5. What is done about unrepresented expression values?
5.1 Using else branch:
default case executed
5.2 No else branch provided:
enumValuePrint: BBBB
```

1.5 PHP

- **What is the form and type of the control expression?**
  In PHP, the control expression in a switch statement is written inside the parentheses following the `switch` keyword. This expression is evaluated once, and its value is compared sequentially against each `case` label. The control expression can take several forms, including variables, literals, function calls, and arithmetic expressions. The language supports various types for the control expression, such as `int`, `float`, `string`, and `bool`. In recent versions of PHP, arrays can also be used in the control expression.

```php
echo "1. What is the form and type of the control expression?\n";
echo "1.1 Forms of control expression:\n";

// variable
$x = 2;
switch ($x) {
    case 1:
        echo "x is 1\n";
        break;
    case 2:
        echo "x is 2\n";
        break;
    default:
        echo "default case executed\n";
}

$y = 3;
switch ($x + $y) {
    case 4:
        echo "x + y is 4\n";
        break;
    case 5:
        echo "x + y is 5\n";
        break;
    default:
        echo "default case executed\n";
}

// literal
switch ("hello") {
    case "hi":
        echo "The string is 'hi'\n";
        break;
    case "hello":
```

```
        echo "The string is 'hello'\n";
        break;
    default:
        echo "default case executed\n";
}

// function call
switch (test()) {
    case 1:
        echo "func returned 1\n";
        break;
    case 2:
        echo "func returned 2\n";
        break;
    default:
        echo "default case executed\n";
    }

echo "1.2 Types of control expression:\n";

// int
$intValue = 3;
switch ($intValue) {
    case 1:
        echo "intValue is 1\n";
        break;
    case 3:
        echo "intValue is 3\n";
        break;
}

// string
$strValue = "kemal";
switch ($strValue) {
    case "mustafa":
        echo "strValue is mustafa\n";
        break;
    case "kemal":
        echo "strValue is kemal\n";
        break;
    default:
        echo "default case executed\n";
}
```

```php
// boolean
$boolValue = true;
switch ($boolValue) {
    case true:
        echo "boolValue is true\n";
        break;
    case false:
        echo "boolValue is false\n";
        break;
}

// float
$floatValue = 1.5;
switch ($floatValue) {
    case 1.0:
        echo "floatValue is 1.0\n";
        break;
    case 1.5:
        echo "floatValue is 1.5\n";
        break;
    default:
        echo "floatValue is neither 1.0 nor 1.5\n";
}

$arrayValue = [1, 2, 3];
switch ($arrayValue) {
    case [1, 2]:
        echo "arrayValue is[1, 2]\n";
        break;
    case [1, 2, 3]:
        echo "arrayValue is [1, 2, 3]\n";
        break;
    default:
        echo "not found\n";
}
```

Printed Output:

```
1. What is the form and type of the control expression?
1.1 Forms of control expression:
x is 2
x + y is 5
The string is 'hello'
```

```
func returned 2
1.2 Types of control expression:
intValue is 3
strValue is kemal
boolValue is true
floatValue is 1.5
arrayValue is [1, 2, 3]
```

- **How are the selectable segments specified?**
  In PHP, each selectable segment in a `switch` statement is defined using the `case` keyword, followed by a value and a colon. The corresponding code block for that case follows, and execution continues until a `break` statement is encountered or the end of the `switch` block is reached. Each `case` segment is separated and typically ends with `break` to prevent fallthrough. These segments can contain any valid PHP code.

```php
echo "\n\n2. How are the selectable segments specified?\n";
$value = 1;
switch ($value) {
    case 1:
        echo "case 1 \n";
        $localVar = "inside case 1";
        echo "Local variable: $localVar\n";
        $value += 10;
        echo "Value after modification: $value\n";
        break;

    case 2:
        echo "case 2 \n";
        break;

    default:
        echo "default case executed\n";
}
```

Printed Output:

```
2. How are the selectable segments specified?
case 1
Local variable: inside case 1
Value after modification: 11
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  In PHP, execution flow within a `switch` statement is not automatically restricted to a single `case` block. By default, once a `case` is matched, execution continues into the following `case` blocks unless a

`break` statement is used to explicitly exit the switch. This behavior can be used for grouping cases but may also lead to unintended results. To restrict execution to a single case, a `break` statement should be put at the end of each `case` block,

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

echo "3.1 Using break: \n";
$testValue = 1;
switch ($testValue) {
    case 1:
        echo "testValue is 1\n";
        $testValue = 2;
        break;
    case 2:
        echo "testValue is 2\n";
        break;
    case 3:
        echo "testValue is 3\n";
        break;
}

echo "3.2 Without using break :\n";
$testValue = 1;
switch ($testValue) {
    case 1:
        echo "testValue is 1\n";
    case 2:
        echo "Continues to case 2\n";
    case 3:
        echo "Continues to case 3\n";
        break;
    case 4:
        echo "This won't be reached because of the break in case 3\n"
           ;
        break;
}
```

Printed Output:

```
3. Is execution flow through the structure restricted to include just a single selectable segment?
3.1 Using break:
testValue is 1
3.2 Without using break :
testValue is 1
```

```
Continues to case 2
Continues to case 3
```

- **How are case values specified?**
  In PHP, `case` values in a `switch` statement can be specified using literals, variables, or expressions. Language supports various value types for cases, including integers, strings, and the result of arithmetic operations. Multiple `case` labels can share the same code block by listing them consecutively without a `break` between them. PHP's `switch` uses loose comparison (`==`) by default when evaluating the control expression against `case` values. This means that different data types may be considered equal if their values are loosely equivalent. Also different types from the control expression can be written in the case values.

```php
echo "\n\n4. How are case values specified?\n";

// literal
$name = "kemal";
switch ("kemal") {
    case "mustafa":
        echo "name is mustafa\n";
        break;
    case "kemal":
        echo "name is kemal\n";
        break;
    default:
        echo "default case executed\n";
}

// variable
switch ("kemal") {
    case "mustafa":
        echo "name is mustafa\n";
        break;
    case $name:
        echo "name is kemal\n";
        break;
    default:
        echo "default case executed\n";
}

// expression
$num = 5;
$num1 = 3;
$num2 = 2;
switch ($num) {
```

```
    case $num1 + $num2:
        echo "num is equal to num1 + num2\n";
        break;
    case $num1 * $num2:
        echo "num is equal to num1 * num2\n";
        break;
    default:
        echo "default case executed\n";
}

// multiple cases sharing the same code block
$number = 1;
switch ($number) {
    case 2:
    case 1:
        echo "Number is either 1 or 2\n";
        break;
    case 3:
        echo "Number is 3\n";
        break;
    default:
        echo "default case executed\n";
}

// different types are allowed in cases
$number = 2;
switch ($number) {
    case 2:
        echo "number is 2\n";
        break;
    case 3:
        echo "number is 3\n";
        break;
    case "irem":
        echo "number is irem\n";
        break;
    default:
        echo "number case executed\n";
}


// loose comparisons
$value = "2"; // String "2"
```

```php
switch ($value) {
    case 2:  // int 2
        echo "loose comparison (==) in switch statements\n";
        break;
    default:
        echo "not reached\n";
}
```

Printed Output:

```
4. How are case values specified?
name is kemal
name is kemal
num is equal to num1 + num2
Number is either 1 or 2
number is 2
loose comparison (==) in switch statements
```

- **What is done about unrepresented expression values?**
  In PHP, if none of the `case` labels match the control expression in a `switch` statement, execution will continue only if a `default` case is provided. When the `default` case is provided, the unrepresented expression values action can be taken here. However, the use of it is not optional.

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

echo "5.1 Using default case:\n";
$number = 4;
switch ($number) {
    case 1:
        echo "number is 1\n";
        break;
    case 2:
        echo "number is 2\n";
        break;
    case 3:
        echo "number is 3\n";
        break;
    default:
        echo "default case executed\n";
}

echo "5.2 No default case provided:\n";
switch ($number) {
    case 1:
        echo "number is 1\n";
```

```
        break;
    case 2:
        echo "number is 2\n";
        break;
    case 3:
        echo "number is 3\n";
        break;
}
```

Printed Output:

```
5. What is done about unrepresented expression values?
5.1 Using default case:
default case executed
5.2 No default case provided:
```

1.6 Python

- **What is the form and type of the control expression?**
  The control expression comes after the `match` keyword. This expression is evaluated once and then tried to match. The control expression can take many forms, such as a variable, literal, function call, or arithmetic operation. The language also supports a wide range of types for the control expression. Primitives ones are the `int`, `string`, `float`, `boolean` ones. Also it can take collections such as `tuple`, `list`, or `dict` or `objects`.

```
print("1. What is the form and type of the control expression?")
print("1.1 Forms of control expression")
# variable
x = 1
match x:
    case 1:
        print("x is 1")
    case 2:
        print("x is 2")
# arithmetic expression
x = 1
y = 2
match x+y:
    case 2:
        print("x+y is 2")
    case 3:
        print("x+y is 3")
# literal value
match "hello":
    case "hello":
        print("hello")
    case _:
        print("not hello")
# function call
def test():
    return 1

match test():
    case 1:
        print("test() is 1")
    case _:
        print("test() is not 1")
print("1.2 Types of the control expression")
# int
x = 1
```

```
print(type(x))
match x:
    case 1:
        print("x is 1")
    case 2:
        print("x is 2")
# str
x = "hello"
print(type(x))
match x:
    case "hello":
        print("x is 'hello'")
    case "world":
        print("x is 'world'")
# bool
x = True
print(type(x))
match x:
    case True:
        print("x is True")
    case False:
        print("x is False")
# float
x = 1.0
print(type(x))
match x:
    case 1.0:
        print("x is 1.0")
    case 2.0:
        print("x is 2.0")
# tuple
x = (1, 2)
print(type(x))
match x:
    case (1, 2):
        print("x is (1, 2)")
    case (2, 1):
        print("x is (2, 1)")
# list
x = [1, 2]
print(type(x))
match x:
    case [1, 2]:
```

```
            print("x is [1, 2]")
        case [2, 1]:
            print("x is [2, 1]")
# dictionary
x = {"a": 1, "b": 2}
print(type(x))
match x:
    case {"a": 1, "b": 2}:
        print("x is {'a': 1, 'b': 2}")
    case {"b": 2, "a": 1}:
        print("x is {'b': 2, 'a': 1}")

# object
class ObjectForMatch:
    def __init__(self, num):
        self.num = num

x = ObjectForMatch(1)
y = ObjectForMatch(2)
z = ObjectForMatch(1)

print(type(x))
print(type(y))
print(type(z))
match x:
    case ObjectForMatch() if x.num == y.num: # if the if part is not
       used, it will give error: Irrefutable pattern is allowed only
       for the last case
        print("x is y")
    case ObjectForMatch() if x.num == z.num:
        print("x is z")
    case _:
        print("default case")
```

Printed Output:

```
1. What is the form and type of the control expression?
1.1 Forms of control expression
x is 1
x+y is 3
hello
test() is 1
1.2 Types of the control expression
<class 'int'>
```

```
x is 1
<class 'str'>
x is 'hello'
<class 'bool'>
x is True
<class 'float'>
x is 1.0
<class 'tuple'>
x is (1, 2)
<class 'list'>
x is [1, 2]
<class 'dict'>
x is {'a': 1, 'b': 2}
<class '__main__.ObjectForMatch'>
<class '__main__.ObjectForMatch'>
<class '__main__.ObjectForMatch'>
x is z
```

- **How are the selectable segments specified?**
  The selectable segments are specified using the `case` keyword. It follows a pattern, a colon (:), and an intended code block. Each segment must be intended as Python enforces indentation. It is possible to define variables, functions, or perform any statements inside a case block similar to inside a function or an if-statement.

```
print("\n\n2. How are the selectable segments specified?")

var = 1
match var:
    case 1:
        print("Matched 1")
        def test():
            return "Inside case 1"

        print(test())
        list1 = [3, 2, 1]
        list1.sort()
        print("Sorted list:", list1)
        var = 3
    case 2:
        print("Matched 2")
    case 3:
        print("Matched 3")
```

Printed Output:

```
2. How are the selectable segments specified?
Matched 1
Inside case 1
Sorted list: [1, 2, 3]
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  Match statement only allows a single selectable segment to be executed. Once a case pattern matches the control expression, the corresponding block runs and the match structure ends. There is no need to `break`. In the example, it is shown that when there is second pattern that is matching, it is not executed.

```
print("\n\n 3. Is execution flow through the structure restricted to
    include just a single selectable segment?")
var = 1
match var:
    case 1:
        print("var is 1")
        var = 3
    case 1:
        print("var is 1 again")
    case 3:
        print("var is 3")
print(var)
```

Printed Output:

```
3. Is execution flow through the structure restricted to include just a single selectable segment?
var is 1
3
```

- **How are case values specified?**
  Case values are specified after the `case` keyword. They can be of various types.
  - **Literal patterns**: Constants as numbers (1, 2.5), strings ("irem"), and booleans (True, False)
  - **Structural patterns**: Data structures such as lists [1, 2], tuples (1, 2), and dictionaries {"a": 1, "b": 2}
  - **Special patterns**: The pattern _ matches anything and used as a default case
  - **Capture patterns**: When a variable name is used directly (like `case var2:`), it acts as an "irrefutable pattern" that always matches and captures the matched value into that variable

  Problematic part of its implementation is that when a variable is directly used, the value in the control expression is copied into the case value, comparison is not done. These "irrefutable patterns" must be the last case since they always match. To compare against a variable's value, it is needed to use a guard clause with `case var if condition:`.

```
print("\n\n 4. How are the case values specified?")
var = 1
var2 = 1
match var:
    case "irem":
        print("var is irem")
    case [1, 2]:
        print("var is [1, 2]")
    case (1, 2):
        print("var is (1, 2)")
    case {"a": 1, "b": 2}:
        print("var is {'a': 1, 'b': 2}")
    case _ if var2 == var:
        print("var is var2")
    case _:
        print("not found")

x = 5
var2 = 10
match x:
    case var2:
        print(f"var2 is {var2}")
```

Printed Output:

```
 4. How are the case values specified?
var is var2
var2 is 5
```

- **What is done about unrepresented expression values?**
  If there is a part of the `case _:` in the match case statement, , it acts as a default case, and the statements under it will be executed when none of the other patterns match the control expression. If the `case _:` block is not provided and no cases match, then nothing is executed and the program continues after the match block.

```
print("\n\n 5. What is done about unrepresented expression values?")
var = 2
match var:
    case 1:
        print("var is 1")
    case _:
        print("not found")

match var:
```

```
    case 1:
        print("var is 1")
```

Printed Output:

```
 5. What is done about unrepresented expression values?
not found
```

1.7 Rust

The below enum and function is used in the following questions:

```
fn test() -> i32 {
    2
}
enum EnumExample {
    Ex1,
    Ex2,
}
```

- **What is the form and type of the control expression?**
  In Rust, the control expression in a 'match' statement can take various forms. It may be a variable, a literal, the result of a function call, or an arithmetic or logical expression. Rust supports a wide range of types as control expressions. These include primitive types such as integers, floating-point numbers, booleans, characters, and strings. More complex types are also supported, including tuples, and enums. Each pattern must be compatible with the control expression's type, and the match arms are checked to ensure all possible values are handled, either explicitly or through a wildcard pattern (_).

```
println!("1. What is the form and type of the control expression?");
println!("1.1 Forms of control expression:");

// variable
let x = 2;
match x {
    1 => println!("x is 1"),
    2 => println!("x is 2"),
    _ => println!("default case executed"),
}

// arithmetic expression
let y = 3;
match x + y {
    4 => println!("x + y is 4"),
    5 => println!("x + y is 5"),
    _ => println!("default case executed"),
}

// literal
match "mustafa" {
    "mustafa" => println!("string is 'mustafa'"),
    "kemal" => println!("tring is 'kemal'"),
    _ => println!("default case executed"),
```

```
}

// function call
match test() {
    1 => println!("func returned 1"),
    2 => println!("func returned 2"),
    _ => println!("default case executed"),
}

println!("1.2 Types of control expression:");

// i32
let int_value: i32 = 3;
match int_value {
    1 => println!("int_value is 1"),
    3 => println!("int_value is 3"),
    _ => println!("default case executed"),
}

// string
let str_value = "mustafa";
match str_value {
    "mustafa" => println!("string is 'mustafa'"),
    "kemal" => println!("string is 'kemal'"),
    _ => println!("default case executed"),
}

// character
let char_value = 'A';
match char_value {
    'A' => println!("char_value is 'A'"),
    'B' => println!("char_value is 'B'"),
    _ => (),
}

// Boolean type
let bool_value = true;
match bool_value {
    true => println!("bool_value is true"),
    false => println!("bool_value is false"),
}

// Tuple type
```

```
let tuple_value = (1, "test");
match tuple_value {
    (1, "test") => println!("tuple_value is (1, \"test\")"),
    (1, _) => println!("tuple_value's first element is 1, second
       element is something else"),
    (_, "test") => println!("tuple_value's first element is something
        else, second element is \"test\""),
    _ => println!("tuple_value is something else entirely"),
}

// Enum type
let direction = EnumExample::Ex1;
match direction {
    EnumExample::Ex1 => println!("EnumExample is Ex1"),
    EnumExample::Ex2 => println!("EnumExample is Ex2"),
}
```

Printed Output:

```
1. What is the form and type of the control expression?
1.1 Forms of control expression:
x is 2
x + y is 5
string is 'mustafa'
func returned 2
1.2 Types of control expression:
int_value is 3
string is 'mustafa'
char_value is 'A'
bool_value is true
tuple_value is (1, "test")
EnumExample is Ex1
```

- **How are the selectable segments specified?**
  In Rust, selectable segments in a match statement are defined using pattern arms, each starts with a pattern (such as a literal, range, or structure), followed by a `=>` symbol and a block of code. The code block may be a single expression or a braced block containing multiple statements. Each arm is separated by a comma. If the block is enclosed in braces, it can contain multiple lines. If one statement is needed in the pattern arm, then the braces is not a must. Importantly, each arm can return a value, and when a match statement is used as an expression, the last expression in the executed arm becomes the return value of the entire match block.

```
println!("\n2. How selectable segments are specified?");

let value = 1;
```

```
match value {
    1 => {
        println!("Case 1 executed");
        let local_var = "inside case 1";
        println!("Local variable: {}", local_var);
    },
    2 => {
        println!("Case 2 executed");
    },
    _ => println!("Default case executed"),
}

let x = 2;
let result = match x {
    1 => "one",
    2 => { println!("matched two"); "two" },
    _ => "other",
};
println!("result: {}", result);
```

Printed Output:

```
2. How selectable segments are specified?
Case 1 executed
Local variable: inside case 1
matched two
result: two
```

- **Is execution flow through the structure restricted to include just a single selectable segment?**
  In Rust, execution within a match statement is restricted to a single matching arm. Once a pattern matches the control expression, only the code block associated with that pattern is executed, and the match expression is exited immediately. There is no fallthrough between arms, and each pattern must be unique—defining the same pattern multiple times results in a warning.

```
println!("\n3. Is execution flow through the structure restricted to
   include just a single selectable segment?");

let test_value = 1;
println!("test_value = {}", test_value);

let result = match test_value {
    1 => {
        println!("case 1 matched");
        // last expression is returned
```

```
        "value was 1"
    },
    2 => {
        println!("case 2 matched");
        "Value was 2"
    },
    1 => {
        println!("1 matched again");
        "Second case for 1"
    }, // warning: unreachable pattern
    _ => "Value was something else",
};

println!("result: {}", result);
```

Printed Output:

```
3. Is execution flow through the structure restricted to include just a single selectable segment?
test_value = 1
case 1 matched
result: value was 1
```

- **How are case values specified?**
  In Rust, case values in a match statement can be specified using different patterns. These patterns can take many forms. Literal patterns are used to match exact values, such as specific numbers or strings. Rust also supports the use of range patterns like 1..=5 to match values within a range, and OR patterns using the — symbol to allow a single arm to match multiple values. In addition, Rust allows match guards, which are conditions introduced with if to add extra logic to a pattern match. If the variables are not used with guards, it causes binding and might lead to unintended results.

```
println!("\n4. How case values are specified:");

// Literal patterns
let nameCase = "mustafa";
match nameCase {
    "mustafa" => println!("Name is 'mustafa'"),
    "kemal" => println!("Name is 'kemal'"),
    _ => println!("Name is neither 'mustafa' nor 'kemal'"),
}

// variables
let variable = 3;
let variable1 = 1;
let variable2 = 3;
let variable3 = 3;
```

```
match variable{
    variable1 => println!("variable is variable1"),
    variable2 => println!("variable is variable2"),
    variable3 => println!("variable is variable3"),
    _ => println!("variable is something else"),
}

// Range patterns
let num = 7;
match num {
    1..=5 => println!("num is between 1 and 5"),
    6..=10 => println!("num is between 6 and 10"),
    _ => println!("num is outside of ranges"),
}

// OR patterns
let orValue = 2;
match orValue {
    2|4|6 => println!("orValue is even"),
    1|3|5 => println!("orValue is odd"),
    _ => println!("other cases"),
}
// guard patterns
let score = 85;
match score {
    s if s > 0 => println!("s is positive"),
    s if s == 0 => println!("s is 0"),
    s if s < 0 => println!("s is negative"),
    _ => println!("default case"),
}

// patterns for tuples
let point = (3, 4);
match point {
    (0, 0) => println!("Origin"),
    (0, y) => println!("On y at y={}", y),
    (x, 0) => println!("On x at x={}", x),
    (x, y) => println!("Point at ({}, {})", x, y),

}
```

Printed Output:

4. How case values are specified:

```
Name is 'mustafa'
variable is variable1
num is between 6 and 10
orValue is even
s is positive
Point at (3, 4)
```

- **What is done about unrepresented expression values?**
  In Rust, unrepresented expression values in a match statement must be handled. The language enforces exhaustive matching, it means every possible value of the control expression must be covered by the arms. If not all values are covered, and no wildcard arm (_) is included, the code will not compile. When it is not practical or necessary to match every specific value, the underscore (_) pattern can be used. It serves the same role as a default case in other languages.

```rust
let name = "mustafa";
match name {
    "mustafa" => println!("name is mustafa"),
    "kemal" => println!("name is kemal"),
    "ataturk" => println!("name is ataturk"),
    _ => println!("name is not mustafa kemal ataturk")
}

// enum example
let enumEx = enumExample::Variant1;
match enumEx {
    enumExample::Variant1 => println!("It is variant 1."),
    enumExample::Variant2 => println!("It is variant 2.")
}
```

Printed Output:

```
5. Handling unrepresented expression values:
name is mustafa
It is variant 1.
```

2 Part B - Evaluation

This section includes evaluation of these languages in terms of readability and writability of multiple-way selection statements.

2.1 Dart

In Dart, the syntax of `switch` statements is generally clear when used in its traditional form, but it becomes less intuitive when used as a `switch` expression, since the syntax shifts from colons and cases to arrows (`=>`). This inconsistency harms writability. However, the implementation of fallthrough is readable as it continues if no statement exists between the cases, but it stops when any statement is executed.

2.2 Go

Go's `switch` statement is readable due to its simple structure and the requirement for explicit `fallthrough` when execution should continue to the next case. This design avoids accidental fallthroughs and makes the flow clear. Go restricts both the control expression and the case values to compatible types, which improves safety and predictability during comparison. Although Go's `switch` is less flexible than in some other languages, it enhances its writability.

2.3 JavaScript

In JavaScript, `switch` statements uses strict equality (`===`) checks, which has both good and bad effects on readability and writability. It improves the reliability of comparisons between the control expression and case values. However, matching objects by reference rather than by value can lead to confusing behavior. Additionally, allowing implicit fallthrough when a `break` is forgotten negatively impacts writability and can cause bugs.

2.4 Kotlin

Kotlin's `when` expression is highly readable, with a clean syntax that uses `->` arrows to associate conditions with results. Smart casting after type checks significantly improves writability by reducing the need for manual type conversions. Kotlin's ability to match on types enhances reliability. Additionally, the enforcement of exhaustiveness in `when` expressions, but not necessarily in simple `when` statements, provides a good balance between flexibility and safety.

2.5 PHP

In PHP, the allowance of implicit fallthrough in `switch` statements harms writability, as forgetting a `break` can lead to unintended behavior. The use of loose comparison (`==`) rather than strict equality can also reduce reliability, when comparing values of different types.

2.6 Python

In Python, the pattern matching feature of `match` statements have significant drawbacks, particularly because variables in case patterns are treated as bindings rather than as values to match, which negatively impacts

readability, writability, and reliability. Python's `match` construct does not support fallthrough, which is positive for clarity and control flow, but might limit the writability. Matching objects can be unreliable and unintuitive, since object comparison relies on pattern rules rather than simple equality.

2.7 Rust

Rust's `match` expression's trong type checking and the enforcement of exhaustive matching improve reliability by ensuring that all possible cases are considered. Being able to use `match` as an expression is also good for writability and readability. .

2.8 Final Decision

Among all the languages evaluated, I find Kotlin's `when` expression the most intuitive one due to its clean syntax. Smart casting after type checks, strong type-based matching, and the balance between exhaustiveness in expressions and flexibility in statements further improves both readability and safety and better language in terms of readability and writability.

3 Part C - Learning Strategy

To run the programs, I used my own computer with local compilers and interpreters. For answering the questions and gaining a general understanding of each language, I primarily consulted the official documentation sources: Dart's documentation [1], Go's documentation [3], JavaScript resources through Mozilla Developer Network [4], Kotlin's documentation [5], PHP's documentation [7], Python's documentation [6], and Rust's documentation [8]. I tried different patterns, structures, and behaviors based on both what felt intuitive and what I observed in other programming languages. When some behaviors or syntax details were unclear or unexpected, I referred to additional sources such as Stack Overflow [9], W3Schools [10], and GeeksforGeeks [2] to clarify specific technical points. I also could not understand whether the if-else should be included to the report, then I consulted to TAs about my confusion.

References

[1] Dart documentation. https://dart.dev/. Accessed: 2025-04-25.

[2] Geeksforgeeks. https://www.geeksforgeeks.org. Accessed: 2025-04-25.

[3] Go documentation. https://go.dev/doc/. Accessed: 2025-04-26.

[4] Javascript — mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed: 2025-04-25.

[5] Kotlin documentation. https://kotlinlang.org/. Accessed: 2025-04-26.

[6] Numpy documentation. https://docs.python.org/3/. Accessed: 2025-04-26.

[7] Php documentation. https://www.php.net/docs.php. Accessed: 2025-04-26.

[8] Rust documentation. https://www.rust-lang.org/learn. Accessed: 2025-04-26.

[9] Stack overflow. https://stackoverflow.com. Accessed: 2025-04-25.

[10] W3schools. https://www.w3schools.com. Accessed: 2025-04-25.