1. B2.1.2 Construct programs that can extract and manipulate substrings

1.1 What is a substring

A substring is a part of a larger string.

Examples

- In `"Computer"`, `"Comp"`, `"put"` and `"er"` are substrings.
- In `"student@school.org"`, `"student"`, `"school.org"` and `"@school"` are substrings.

In Java, strings are objects of the `String` class. You use methods of this class to identify, extract and manipulate substrings.

For this section we will only use direct string methods, not loops.

Key methods

- `length()`
- `charAt(int index)`
- `indexOf(String s)` or `indexOf(char c)`
- `lastIndexOf(String s)`
- `substring(int beginIndex)`
- `substring(int beginIndex, int endIndex)`
- `replace(String old, String new)`

Indices start at zero.

1.2 Extracting substrings with `substring`

Basic forms

```
String sub1 = text.substring(start, end);   // from start up to end - 1

String sub2 = text.substring(start);        // from start to the end
```

Example 1 split a subject name

```
public class SubstringExample1 {

    public static void main(String[] args) {

        String text = "Computer science";


        String word1 = text.substring(0, 8);  // "Computer"

        String word2 = text.substring(9);     // "science"


        System.out.println(word1);

        System.out.println(word2);

    }

}
```

Trace table

Step	text	word1 before	word2 before	Operation	word1 after	word2 after	Output
1	"Computer science"	-	-	`word1 = text.substring(0, 8)`	"Computer"	-	
2	"Computer science"	"Computer"	-	`word2 = text.substring(9)`	"Computer"	"science"	

3	"Computer science"	"Computer"	"science"	`println(word1)`			"Computer"
4	"Computer science"	"Computer"	"science"	`println(word2)`			"science"

1.3 Using `indexOf` to find positions

Often you need to find where a substring begins.

```
int position = text.indexOf("@");
```

If the substring is found, `indexOf` returns its index. If not, it returns `-1`.

Example 2 get the domain from an email address

```
public class SubstringExample2 {

    public static void main(String[] args) {

        String email = "student@school.org";


        int atPos = email.indexOf("@");

        String user = email.substring(0, atPos);

        String domain = email.substring(atPos + 1);


        System.out.println("User: " + user);

        System.out.println("Domain: " + domain);
```

```
    }
}
```

Trace table

Step	email	atPos	user before	domain before	Operation	user after	domain after	Output
1	"student@school.org"	-	-	-	`atPos = email.indexOf("@")`	-	-	
2	"student@school.org"	7	-	-	`user = email.substring(0, atPos)`	"student"	-	
3	"student@school.org"	7	"student"	-	`domain = email.substring(8)`	"student"	"school.org"	
4					print user			"User: student"
5					print domain			"Domain: school.org"

1.4 Altering substrings with `replace`

You can alter part of a string by replacing one substring with another.

```
String newText = oldText.replace("old", "new");
```

Example 3 change a subject name

```
public class SubstringExample3 {

    public static void main(String[] args) {

        String sentence = "I love maths";

        String updated = sentence.replace("maths", "computer
science");


        System.out.println(sentence);

        System.out.println(updated);

    }

}
```

Trace table

Step	sentence	updated before	Operation	updated after	Output
1	"I love maths"	-	`updated = sentence.replace("maths", ...)`	"I love computer science"	
2	"I love maths"	"I love computer science"	`println(sentence)`		"I love maths"

3	"I love maths"	"I love computer science"	`println(updated)`		"I love computer science"

1.5 Concatenating substrings

You join strings using the `+` operator.

Example 4 format a full name

```
public class SubstringExample4 {

    public static void main(String[] args) {

        String fullName = "Ada Lovelace";


        int spacePos = fullName.indexOf(" ");

        String first = fullName.substring(0, spacePos);

        String last = fullName.substring(spacePos + 1);


        String formatted = last + ", " + first;


        System.out.println(formatted);

    }

}
```

Here you see:

- `indexOf` to find the space
- `substring` to extract first and last name
- `+` to join them in a new format

Program construction without using string functions.

1. Counting vowels in a string

1.1 Program

```
public class CountVowels {

    public static void main(String[] args) {

        String text = "International Baccalaureate";

        int vowelCount = 0;


        // convert to lower case so we do not worry about upper case
letters

        text = text.toLowerCase();


        for (int i = 0; i < text.length(); i++) {

            char c = text.charAt(i);


            if (c == 'a' || c == 'e' || c == 'i'

                    || c == 'o' || c == 'u') {

                vowelCount = vowelCount + 1;

            }

        }
```

```
        System.out.println("Number of vowels: " + vowelCount);
    }
}
```

1.2 Explanation

• We loop from index zero to index `length minus one`.
• For each character, we check if it is one of the five vowels.
• If yes, we increase `vowelCount` by one.
• At the end, we print the total.

1.3 Trace table for the first few characters

Text is `"international"`, converted to lower case (already lower).

Iteration	i	c = text.charAt(i)	Is c a vowel	vowelCount before	vowelCount after
1	0	'i'	yes	0	1
2	1	'n'	no	1	1
3	2	't'	no	1	1
4	3	'e'	yes	1	2
5	4	'r'	no	2	2

And so on until the loop finishes.

---

2. Reversing a string using a loop

2.1 Program

```
public class ReverseString {

    public static void main(String[] args) {

        String text = "Computer Science";

        String reversed = "";


        for (int i = text.length() - 1; i >= 0; i--) {

            char c = text.charAt(i);

            reversed = reversed + c;

        }


        System.out.println("Original: " + text);

        System.out.println("Reversed: " + reversed);

    }

}
```

2.2 Explanation

• We start from the last index `text.length() minus one`.
• Move backward until index zero.
• At each step, we take the character at position `i` and append it to `reversed`.
• The new string is the original text in reverse order.

2.3 Trace table for `"CAT"`

Iteration	i	c = text.charAt(i)	reversed before	reversed after
1	2	'T'	""	"T"
2	1	'A'	"T"	"TA"
3	0	'C'	"TA"	"TAC"

---

3. Removing spaces from a string

3.1 Program

```
public class RemoveSpaces {

    public static void main(String[] args) {

        String text = "IB Computer Science HL";

        String withoutSpaces = "";


        for (int i = 0; i < text.length(); i++) {

            char c = text.charAt(i);


            if (c != ' ') {

                withoutSpaces = withoutSpaces + c;

            }

        }
```

```
        System.out.println("Original: " + text);
        System.out.println("Without spaces: " + withoutSpaces);
    }
}
```

3.2 Explanation

• We loop over every character in `text`.
• If the character is not a space, we append it to `withoutSpaces`.
• If it is a space, we simply skip it.
• The result is the same text but with all spaces removed, for example `"IBComputerScienceHL"`.

---

4. Building our own substring function using a loop

This example shows how to implement a simple version of `substring` using a loop. This helps students understand what the library method does conceptually.

4.1 Method

```
public class ManualSubstring {
    public static String mySubstring(String text, int start, int end)
{
        String result = "";

        for (int i = start; i < end; i++) {
            char c = text.charAt(i);
            result = result + c;
```

```
        }

        return result;
    }

    public static void main(String[] args) {
        String text = "International";
        String part = mySubstring(text, 2, 7);  // expected "terna"
        System.out.println(part);
    }
}
```

4.2 Explanation

- `start` and `end` work in the same way as Java `substring` (end is exclusive).
- We initialise `result` as an empty string.
- The loop runs from `start` to `end minus one`.
- At each step we append the current character to `result`.
- After the loop, `result` contains the required substring.

4.3 Trace table for `"HELLO"`, `start 1`, `end 4`

Iteration	i	c = text.charAt(i)	result before	result after
1	1	'E'	""	"E"
2	2	'L'	"E"	"EL"

3	3	'L'	"EL"	"ELL"

Loop stops when `i` reaches 4. Return value is `"ELL"`.

---

5. Replacing all occurrences of one character using a loop

Here we manually replace one character with another, similar in effect to `replace`.

5.1 Program

```
public class ManualReplaceChar {

    public static void main(String[] args) {

        String text = "banana";

        char target = 'a';

        char replacement = 'o';


        String result = "";


        for (int i = 0; i < text.length(); i++) {

            char c = text.charAt(i);


            if (c == target) {

                result = result + replacement;

            } else {
```

```
                result = result + c;
            }
        }

        System.out.println("Original: " + text);
        System.out.println("Changed: " + result);
    }
}
```

5.2 Explanation

• We read each character from the original string.
• If this character equals `target`, we place `replacement` into the result.
• Otherwise we keep the original character.
• The final string has all `a` changed to `o`, in this case `"bonono"`.

---

6. Counting occurrences of a substring pattern (simple version)

This example demonstrates using a loop to count how many times `"ab"` appears in a string. We still avoid `indexOf` in the core logic.

6.1 Program

```
public class CountPattern {

    public static void main(String[] args) {

        String text = "abababxaba";

        String pattern = "ab";
```

```
        int count = 0;


        for (int i = 0; i <= text.length() - pattern.length(); i++) {
            char c1 = text.charAt(i);
            char c2 = text.charAt(i + 1);

            if (c1 == 'a' && c2 == 'b') {
                count = count + 1;
            }
        }

        System.out.println("Number of 'ab' patterns: " + count);
    }
}
```

6.2 Explanation

• We want to compare pairs of characters.
• We loop until index `length minus pattern length` so that `i + 1` stays inside the string.
• For each index `i`, we check if `text[i]` is `a` and `text[i+1]` is `b`.
• If both checks are true, we increase the counter.

---

2. B2.1.3 How programs use common exception handling techniques

2.1 Potential points of failure

Programs can fail at several points. The syllabus requires three categories.

1. Unexpected inputs
   • User types letters where a number is expected
   • User enters an empty string

2. Resource unavailability
   • File does not exist or cannot be opened
   • Network resource cannot be reached

3. Logic errors
   • Division by zero
   • Array index out of range
   • Using a variable or reference that is not valid

Without exception handling, such problems often cause the program to stop with an error message and a stack trace.

2.2 Role of exception handling in developing programs

Exception handling allows a programmer to

• catch errors at run time
• respond in a controlled way instead of crashing
• notify the user with a clear message
• close resources such as files and network connections
• keep the program stable or shut it down cleanly

In Java, the basic pattern uses `try`, `catch`, and `finally`.

Python has similar structures called `try`, `except`, and `finally`.

2.3 Basic `try catch finally` in Java

General pattern

```
try {
```

```
    // code that may throw an exception
} catch (ExceptionType e) {
    // what to do if that exception happens
} finally {
    // code that always runs
}
```

There can be several `catch` blocks for different exception types, but there is at most one `finally`.

2.4 Example 1 handling invalid number input

```
import java.util.Scanner;


public class ExceptionExample1 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);


        try {
            System.out.print("Enter an integer: ");
            String text = sc.nextLine();


            int number = Integer.parseInt(text);
            System.out.println("You entered: " + number);
```

```
        } catch (NumberFormatException e) {
            System.out.println("That was not a valid integer.");
        } finally {
            System.out.println("Input attempt finished.");
            sc.close();
        }
    }
}
```

Explanation

• Possible unexpected input is non numeric text
• Inside `try`, `Integer.parseInt` may throw `NumberFormatException`
• `catch` handles this specific error and prints a friendly message
• `finally` runs whether or not the error occurred, so the scanner is always closed and a final message is printed

Trace table for input `"42"`

Step	text	number before	Exception?	Output
1	"42"	-	no	prompt: "Enter an integer: "
2		42	no	"You entered: 42"
3		42	no	"Input attempt finished."

Trace table for input `"abc"`

Step	text	number before	Exception?	Output
1	"abc"	-	`NumberFormatException`	prompt: "Enter an integer: "
2		-	yes, caught	"That was not a valid integer."
3		-	handled	"Input attempt finished."

2.5 Example 2 handling division by zero

This is a logic error that can be caught with a `catch` block.

```
public class ExceptionExample2 {

    public static void main(String[] args) {

        int numerator = 10;

        int denominator = 0;


        try {

            int result = numerator / denominator;

            System.out.println("Result is " + result);

        } catch (ArithmeticException e) {

            System.out.println("Cannot divide by zero.");

        } finally {

            System.out.println("Computation attempt complete.");
```

```
        }
    }
}
```

Here the `try` block contains a potential division by zero.
If `denominator` is zero, Java throws `ArithmeticException`, which is caught.

2.6 Example 3 simple file access with error handling

Even without going deep into file classes, you can show the general idea.

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

public class ExceptionExample3 {
    public static void main(String[] args) {
        BufferedReader reader = null;

        try {
            reader = new BufferedReader(new FileReader("data.txt"));
            String line = reader.readLine();
            System.out.println("First line: " + line);
        } catch (IOException e) {
            System.out.println("There was a problem reading the
file.");
```

```
        } finally {
            try {
                if (reader != null) {
                    reader.close();
                }
            } catch (IOException e) {
                System.out.println("Could not close the file.");
            }
        }
    }
}
```

Possible failure

• Resource unavailability: file missing or not readable

Exception handling

• `IOException` in the `try` block is caught
• `finally` ensures the file is closed if it was opened

---

3. B2.3.3 Looping structures and repeated actions

3.1 Types of loops

Two main categories

1. Counted loops
    • Used when you know in advance how many times to repeat

- In Java, this is usually a `for` loop

2. Conditional loops
   - Used when you do not know the number of repetitions
   - Loop continues while a condition is true
   - In Java, `while` and `do while` are conditional loops

Loops often use conditional statements inside them, with Boolean and relational operators, to control behaviour during repetition.

3.2 Counted loops with `for`

General form

```
for (initialization; condition; update) {

    // repeated statements

}
```

The loop runs as long as the condition is true, and updates the loop variable after each iteration.

Example 4 sum of first five natural numbers

```
public class LoopExample1 {

    public static void main(String[] args) {

        int sum = 0;


        for (int i = 1; i <= 5; i++) {

            sum = sum + i;

        }
```

```
        System.out.println("Sum is " + sum);
    }
}
```

Trace table

Iteration	i before condition	Condition `i <= 5`	sum before	sum after	i after update
1	1	true	0	1	2
2	2	true	1	3	3
3	3	true	3	6	4
4	4	true	6	10	5
5	5	true	10	15	6
6	6	false	15	15	-

Loop stops when `i` becomes 6 and the condition is false. The program prints `"Sum is 15"`.

3.3 Conditional loops with `while`

A `while` loop repeats while its condition is true.

```
while (condition) {
    // repeated statements
```

```
}
```

Example 5 input validation loop

This uses a conditional loop and selection inside the loop.

```
import java.util.Scanner;


public class LoopExample2 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);


        int number = -1;


        while (number < 0 || number > 10) {
            System.out.print("Enter a number from 0 to 10: ");
            number = sc.nextInt();
        }


        System.out.println("You entered: " + number);
        sc.close();
    }
}
```

Explanation

• Loop repeats while the number is outside the range 0 to 10
• Condition uses relational operators `<` and `>` and the logical operator `||` (OR)
• When the user finally enters a valid number, the condition becomes false and the loop ends

Trace table for user inputs 15, then 12, then 7

| Loop pass | number before condition | Condition `number < 0 || number > 10` | User input | number after input | Loop continues |

|---|---|---|---|---|---|
1	-1	true	15	15	yes
2	15	true	12	12	yes
3	12	true	7	7	yes
4	7	false	-	7	no, exit

3.4 Conditional statements inside loops

It is common to place `if` statements inside loops.

Example 6 count even numbers between 1 and 10

```
public class LoopExample3 {

    public static void main(String[] args) {

        int countEven = 0;


        for (int i = 1; i <= 10; i++) {

            if (i % 2 == 0) {

                countEven = countEven + 1;

            }

        }


        System.out.println("Number of even values is " + countEven);
```

```
    }
}
```

This loop

- uses a counted loop from 1 to 10
- uses a relational test `i % 2 == 0` inside an `if`
- uses a Boolean expression to decide whether to update `countEven`

Partial trace table

Iteration	i before condition	Condition `i <= 10`	Test `i % 2 == 0`	countEven before	countEven after
1	1	true	false	0	0
2	2	true	true	0	1
3	3	true	false	1	1
4	4	true	true	1	2
5	5	true	false	2	2
6	6	true	true	2	3
7	7	true	false	3	3
8	8	true	true	3	4
9	9	true	false	4	4

10	10	true	true	4	5
11	11	false	not evaluated	5	5

3.5 `do while` loop

A `do while` loop executes the body once before checking the condition. This is useful when at least one execution is always required.

```
do {
    // statements
} while (condition);
```

Example 7 repeat menu at least once

```
import java.util.Scanner;


public class LoopExample4 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int choice;

        do {
            System.out.println("1. Print Hello");
            System.out.println("2. Print Goodbye");
            System.out.println("3. Exit");
```

```
            System.out.print("Enter choice: ");
            choice = sc.nextInt();

            if (choice == 1) {
                System.out.println("Hello");
            } else if (choice == 2) {
                System.out.println("Goodbye");
            }
        } while (choice != 3);

        sc.close();
    }
}
```

The loop continues as long as `choice` is not 3.
The conditional statement inside the loop uses relational operators and selection.

---

4. IB style practice questions

Use IB command terms such as define, describe, construct, explain, trace, discuss.

4.1 Questions on B2.1.2 substrings

1. **Define** the term substring in the context of Java strings. [2]
2. The variable `email` contains the value `"student@school.org"`.
    a. **Identify** the substring that represents the user name. [1]

b. **Construct** a Java statement that uses `substring` to extract the user name part. [3]

3. **Explain** how the methods `indexOf` and `substring` can be used together to extract the domain portion of an email address in Java. [4]

4. **Construct** a Java method `formatName` that receives a string in the format `"First Last"` and returns a string in the format `"Last, First"`. You may assume that there is exactly one space character separating the first and last names. [6]

A student writes the following code.

```
String sentence = "I love maths";

sentence.replace("maths", "computer science");

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

5. a. **State** the output of this code. [1]
   b. **Describe** why the output is not `"I love computer science"`. [3]
   c. **Construct** a corrected version of the code so that the output is `"I love computer science"`. [3]

4.2 Questions on B2.1.3 exception handling

6. **Describe** three potential points of failure in a Java program that reads an integer from the user and then divides 100 by that number. [4]

7. **Explain** the role of the `finally` block in Java exception handling. In your answer, refer to the management of resources. [4]

8. **Construct** a Java code fragment that

   • reads a line of input from the user as a string
   • attempts to convert it to an integer
   • catches `NumberFormatException` and prints a clear error message
   • always prints `"End of processing"` in a `finally` block

   Do not write the complete class. [6]

A programmer adds `try catch` blocks around all file access code but leaves the `catch` block empty, as shown below.

```
try {

    // file access code

} catch (IOException e) {

}
```

9. **Discuss** the benefits and drawbacks of this approach to exception handling. [6]

4.3 Questions on B2.3.3 loops

10. **Define** a counted loop and a conditional loop, giving one example of each using Java syntax. [4]

Consider the following code.

```
int sum = 0;

for (int i = 1; i <= 4; i++) {

    sum = sum + i;

}

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

11. a. **Construct** a trace table to show the values of `i` and `sum` during each iteration of the loop. [4]
    b. **State** the output of this program. [1]

Explain how Boolean and relational operators are used together in the condition of the following `while` loop.

```
while (number < 0 || number > 10) {
```

```
    // input code
}
```

12. In your answer, you should refer to the effect on the loop execution. [4]

13. **Construct** a Java method that uses a counted loop to print all multiples of 3 from 3 to 30 inclusive, one per line. [5]

14. **Construct** a Java program that uses a conditional `while` loop to repeatedly read integers from the user until the user enters a negative number. The program should then print the count of positive numbers entered. [6]

15. A student writes a `while` loop that never terminates.

    Discuss how careful design of loop conditions, the use of trace tables, and step by step execution in a debugger can help prevent and resolve infinite loops in Java programs. [6]