B2.1.1 Global and local variables and data types in Java

1. What is a variable in Java

A variable is a named box in memory that stores a value.
Every Java variable has

- a type
- a name
- a value
- a scope, which is where in the code it can be used

Examples

```
int age = 16;           // integer

double price = 19.99;   // decimal number

char grade = 'A';       // single character

boolean passed = true;  // true or false

String name = "Raj";    // string
```

Java is statically typed, which means the type is fixed when you declare the variable and you cannot store a different kind of value in it later. For example, if `age` is an `int`, you cannot assign `"sixteen"` to it.

---

2. Required data types

The syllabus here focuses on

- Boolean
- char
- integer
- decimal
- String

Boolean

Represents truth values.

```
boolean isLoggedIn = false;


if (isLoggedIn) {
    System.out.println("Welcome back");
} else {
    System.out.println("Please log in");
}
```

Boolean variables are used mainly for conditions in `if` statements and loops.

char

Represents one character.

```
char letter = 'Z';
char digit = '7';
char symbol = '?';
```

A `char` uses single quotes, and Java stores it as a numeric code under the hood (Unicode).

Integer

Represents whole numbers.

```
int students = 25;
int temperature = -3;
```

Integers are used for counting, indexing arrays, loop counters and so on.

Decimal (floating point)

In Java you usually use `double` for decimal numbers in IB level programs.

```
double average = 82.5;

double pi = 3.14159;
```

Decimal values are used where fractions are possible, like averages, distances or percentages.

String

Represents a sequence of characters.

```
String greeting = "Hello, world";

String id = "CS2027";
```

Strings are objects in Java. They store text, such as user input, messages, file names and so on.

---

3. Local variables in Java

A local variable is declared inside a method, a constructor or a block such as an `if` or a loop.

Properties

- It only exists while the method or block is running
- It can only be used inside that block

Example

```
public static void greet() {

    String message = "Hello student";  // local variable

    System.out.println(message);

}
```

Here

• `message` is local to the `greet` method
 • It is created when `greet` is called
 • It disappears when `greet` finishes

Trying to use `message` outside the method will cause a compile time error.

```
public static void main(String[] args) {

    greet();

    // System.out.println(message); // This does not compile

}
```

---

4. Fields and shared variables in Java

Java does not really have global variables in the same way as some other languages, but it has fields that can be shared by many methods.

A field is a variable declared inside a class but outside any method.

If the field is `static`, all instances of that class share the same copy. This is often used like a global variable in simple IB programs.

```
public class Counter {

    static int count = 0;   // field accessible from all methods in
this class



    public static void increment() {

        count = count + 1;  // uses the shared field

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

        increment();

        increment();

        System.out.println(count); // prints 2

    }

}
```

In this example

• `count` is visible in every static method of the class
• `increment` changes the shared state

Design note for students

• Fields that are shared make some programs simpler
• But too many shared variables can cause bugs when many methods change the same data
• For assessments, you should be able to trace how these fields change as methods are called

---

5. Tracing a Java program with local and shared variables

Consider this Java code.

```
public class Example {

    static int total = 5;  // shared field


    public static void update(int value) {

        int local = value * 2;  // local variable

        total = total + local;

        System.out.println("local is " + local + ", total is " +
total);
```

```
    }


    public static void main(String[] args) {

        update(3);

        update(1);

        System.out.println("final total is " + total);

    }

}
```

Trace table for execution of `main`

Step	Method	value	local	total before	total after	Output
1	main			5	5	
2	update	3	6	5	11	`local is 6, total is 11`
3	update	1	2	11	13	`local is 2, total is 13`
4	main			13	13	`final total is 13`

Local variable `local` is recreated each time `update` runs and does not exist outside that method.
Field `total` preserves its value between method calls.

This style of tracing is what students are expected to do in IB questions that use the command term trace.

B2.1.2 Construct programs that extract and manipulate substrings in Java

Strings are objects. The `String` class provides many methods for substring work.

1. Indexing and positions

Characters in a Java string are numbered starting from zero.

For `"HELLO"`

- index 0 is `H`
- index 1 is `E`
- index 4 is `O`

You can get a character with `charAt`.

```java
String word = "HELLO";

char first = word.charAt(0);   // 'H'

char last = word.charAt(word.length() - 1);  // 'O'
```

2. Extracting substrings with `substring`

There are two main forms:

```java
String sub1 = text.substring(startIndex, endIndex);

String sub2 = text.substring(startIndex);
```

Important rules

- `startIndex` is inclusive
- `endIndex` is exclusive
- indices must be from zero to length

Example

```java
String text = "Computer science";


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

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

3. Finding positions with `indexOf` and `lastIndexOf`

`indexOf` finds the first position where a substring occurs.

```java
String email = "student@school.org";

int atPos = email.indexOf("@");   // position of the at symbol
```

`lastIndexOf` finds the last occurrence, useful for file extensions.

```java
String file = "notes.final.version.docx";

int dotPos = file.lastIndexOf(".");

String extension = file.substring(dotPos + 1);  // "docx"
```

---

4. Common substring tasks

Extract domain name from an email

Task
 Input string such as `"student@school.org"`
 Output `"school.org"`

```java
public static String getDomain(String email) {

    int atPos = email.indexOf("@");
    return email.substring(atPos + 1);

}


public static void main(String[] args) {

    System.out.println(getDomain("student@school.org"));

}
```

Explanation

• `indexOf("@")` returns the index of `@`
• `substring(atPos + 1)` takes everything after that position

Invert name format

Task
Input `"Ada Lovelace"`
Output `"Lovelace, Ada"`

```
public static String invertName(String fullName) {

    int spacePos = fullName.indexOf(" ");

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

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

    return last + ", " + first;

}
```

Step by step for `"Ada Lovelace"`

• `spacePos` becomes 3
• `first` becomes `"Ada"` from index 0 to 2

• `last` becomes `"Lovelace"` from index 4 to end
• Return `"Lovelace, Ada"`

---

5. Manipulating substrings

Replace part of a string

Use `replace` to alter part of a string.

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

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

System.out.println(newSentence); // "I love computer science"
```

Concatenate substrings

Join strings using `+`.

```
String part1 = "Java";

String part2 = "programming";

String combined = part1 + " " + part2;  // "Java programming"
```

You can also build strings from extracted pieces.

```
String code = "IBDP-CS-2027";

String subject = code.substring(5, 7);   // "CS"

String year = code.substring(8);         // "2027"

String result = subject + " " + year;    // "CS 2027"
```

B2.1.3 How programs use exception handling in Java

1. Where can programs fail

The syllabus asks you to know about

• unexpected inputs
• resource unavailability
• logic errors

Unexpected inputs

Examples

• User enters letters when a number is expected
• User gives an empty string for a required value

This often causes `NumberFormatException` or similar errors.

```
String input = "abc";

int n = Integer.parseInt(input);  // NumberFormatException
```

Resource unavailability

Examples

• File does not exist or cannot be opened
• Network connection fails
• Database server is down

These often cause `IOException` or related exceptions.

Logic errors

Examples

• Dividing by zero causes `ArithmeticException`
• Accessing `array[10]` when the array has length 5 causes `ArrayIndexOutOfBoundsException`
• Using a reference that is `null` causes `NullPointerException`

---

2. Purpose of exception handling

Exception handling allows you to

- catch run time errors
- respond in a controlled way
- avoid complete program crashes
- show helpful messages to users
- clean up resources, like closing files

Instead of a long error trace, your program can print a simple message and continue or exit neatly.

---

3. Java `try catch finally` structure

Basic pattern

```
try {

    // code that might throw an exception

} catch (SpecificException e) {

    // what to do if that exception happens

} finally {

    // code that always runs

}
```

You can also have multiple `catch` blocks for different types of exceptions.

```
try {

    // risky operations

} catch (IOException e) {

    // handle input or output problem
} catch (NumberFormatException e) {
    // handle invalid number format
} finally {
    // cleanup code
}
```

---

4. Example 1 File reading with exception handling

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


public class FileExample {
    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("Error reading file: " +
e.getMessage());
```

```
        } finally {
            try {
                if (reader != null) {
                    reader.close();
                }
            } catch (IOException e) {
                System.out.println("Error closing file");
            }
        }
    }
}
```

Explanation

• The `try` block contains code that opens and reads a file, which may fail
• If there is any input output error, the `catch` block runs and prints a clear message
• The `finally` block runs even if there was an error, and attempts to close the file

This pattern protects the program from crashing if the file is missing or unreadable.

---

5. Example 2 Handling user input errors

```
import java.util.Scanner;


public class InputExample {
    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);   // can throw
NumberFormatException
            int result = 10 / number;              // can throw
ArithmeticException
            System.out.println("Result is " + result);
        } catch (NumberFormatException e) {
            System.out.println("You did not enter a valid integer.");
        } catch (ArithmeticException e) {
            System.out.println("Division by zero is not allowed.");
        } finally {
            System.out.println("Thank you for using this program.");
            sc.close();
        }
    }
}
```

Here you can see handling of both unexpected input and a logic error, with user friendly messages.

B2.1.4 Debugging techniques in Java

Debugging is the process of finding and fixing errors. The syllabus expects you to know and use these:

- trace tables
- breakpoint debugging
- print statements
- step by step code execution

1. Trace tables

A trace table is a paper based method to follow the values of variables as a program runs.

Example program

```
public static void main(String[] args) {

    int total = 0;

    for (int i = 1; i <= 3; i++) {

        total = total + i;

    }

    System.out.println(total);

}
```

Trace table

Step	**i**	**total before**	**total after**
1	1	0	1
2	2	1	3
3	3	3	6

So the program prints `6`.

Trace tables are helpful for

• understanding loops
• checking update equations
• spotting mistakes like using `<` instead of `<=` in loop conditions

---

2. Print statements for debugging

One of the simplest debugging techniques is to insert temporary `System.out.println` statements to see what the program is doing.

```
public static int findMax(int[] values) {

    int max = values[0];

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

        System.out.println("Checking index " + i + " value " +
values[i] + " current max " + max);

        if (values[i] > max) {

            max = values[i];

            System.out.println("New max is " + max);

        }

    }

    return max;

}
```

These prints help you see

• the sequence of values
• when the `if` condition is true
• whether the program is entering the correct branch

Students should remember to remove or comment out these debug prints after fixing the bug.

3. Breakpoint debugging

In an integrated development environment such as BlueJ, IntelliJ or Eclipse, you can set a breakpoint on a line of code.

When you run the program in debug mode

• execution pauses at the breakpoint
• you can examine variables in a debug window
• you can execute the code step by step

This allows you to see the exact flow of control and how data changes, without filling the program with print statements.

Even though the exam does not require you to use a specific tool, you should understand the concept of breakpoints and why they are powerful.

4. Step by step execution

Most Java debuggers offer these actions

• step into which moves into a method call line
• step over which runs a method call but does not enter its body in the debugger
• step out which finishes the current method and returns to the caller

These let you control the level of detail when following the program.

For example

If you trust Java library methods such as `substring`, you might step over calls to them.
If you suspect your own method `calculateAverage` is faulty, you would step into it and watch how it updates local variables.

5. Using techniques together

A sensible debugging process in Java might be

1. Reproduce the bug with a clear test case

2. Insert print statements near the suspicious area
3. If the problem is still not obvious, set a breakpoint before that area and run in debug mode
4. Step through line by line, watching variable values
5. Use a trace table for any complex loop or algorithm
6. Once fixed, remove or comment out debug prints and run the test again

---

IB style questions Java focused

You can use these as class work, homework or assessments.

Questions for B2.1.1 Data types and scope

1. **Define** the term local variable as used in Java. [2]
2. **Describe** the difference between a field and a local variable in Java. In your answer, refer to scope and lifetime. [3]
3. **Explain** why a Java program might use a `static` field instead of passing a variable as a parameter to every method that needs it. [4]

Consider the following code.

```
public class Test {

   static int count = 1;



   public static void methodA() {

       int count = 5;

       count = count + 2;

       System.out.println("methodA count: " + count);
```

```
    }

    public static void methodB() {
        count = count + 3;
        System.out.println("methodB count: " + count);
    }

    public static void main(String[] args) {
        methodA();
        methodB();
        System.out.println("main count: " + count);
    }
}
```

4. a. **State** the output of the program. [3]
   b. **Explain** why the value of `count` inside `methodA` is different from the value printed in `methodB`. [4]

Questions for B2.1.2 Substrings

5. **Identify** the substring that should be extracted from `"student@school.org"` to obtain only the domain name. [1]

6. **Construct** a Java method `getExtension` that receives a file name such as `"report.pdf"` and returns the substring representing the file extension, for example `"pdf"`. Use `lastIndexOf` and `substring`. [5]

7. The string `code` has the value `"IBDP-CS-2027"`.

a. **Outline** how a programmer could use `substring` to extract the year part `"2027"`. [2]
b. **Explain** how `indexOf` could be used to make this extraction work even if the subject code changes length. [4]

8. **Construct** a Java method `formatName` which takes a string in the format `"First Last"` and returns `"Last, First"`. Assume there is exactly one space between the two names. [5]

Questions for B2.1.3 Exception handling

9. **Describe** two situations in which a Java program that reads numbers from the keyboard might generate a run time exception. For each situation, name the exception type. [4]

10. **Explain** the role of the `finally` block in Java exception handling. Your answer should refer to resources such as files or network connections. [3]

11. **Construct** a Java code segment that

    • reads a string from the user
    • attempts to convert it to an integer
    • handles invalid input using a `catch` block for `NumberFormatException`
    • always prints `"Input attempt finished"` in a `finally` block.

    Do not write a complete class. [6]

12. A student writes a Java program that tries to open a file and then process its contents. When the file is missing, the program terminates with an uncaught `IOException` and prints a large error trace.

    Discuss how adding appropriate `try catch finally` constructs could improve the user experience and the reliability of the program. [6]

Questions for B2.1.4 Debugging techniques

13. **List** three debugging techniques that can be used when a Java program compiles but produces incorrect output. [3]

14. **Explain** how a trace table can reveal an error in the loop condition of a `for` loop. [4]

Consider the following code.

```
public static void main(String[] args) {

    int product = 1;

    for (int i = 2; i <= 4; i++) {

        product = product * i;

    }

    System.out.println(product);

}
```

15. a. **Construct** a trace table showing the values of `i` and `product` at each step of the loop. [4]
    b. **State** the value printed by the program. [1]

16. **Compare** the use of print statements and breakpoint debugging as methods for finding errors in a Java program. In your answer, give one advantage of each technique. [4]

17. Many novice programmers rely only on print statements for debugging.

    Evaluate this approach in the context of a large Java project developed by a team. [6]