Java-CS-P1

PrimDatTyp

INTE: By-Sh-Int-L
INTE: 8-16-32-64
GKZ: Flo(f)-Do(L)
GKZ: 32-64
INT: Char-16-UniC
VAL: Boolean-1
Unsigned: Non-negative

Unicode Char

\u0000
\uFFFF
2 Bytes cuz': 16-UTF
ASCII: For English

Why Unicode?

Universal for all languages
Includes languages w/ weird letters

Value and Variable

OvFlo: If value > limit

Initialize Var

```
byte Num = 4;
short Num = 21311;
int Num = 62;
long Num = 3333L;
float Num = 3.14f;
double Num = 6.22d;
char a = 'A';
boolean b = true;
```

Valid Var

```
double Num = 3.14;
Use d if bigger than int limit

float Num = .14f;
float Num = 4E2f;
E2 --> 10^2

char a = '\u0041';
Output: a
int unicode = a;
Output: 65

Short s = Short.MAX_VALUE;
Output: Max value of Short
```

Conversion

Konvertierung bedeutet die Umwandlung eines Wertes von einem Datentyp in einen anderen. Dies ist notwendig, wenn Daten verschiedener Typen miteinander verarbeitet werden sollen.

Conversion Types Impl-Expl

IMPL: Byt-Sh-Int-L-Fl-Do
EXPL: Do-Fl-L-Int-Sh-Byte

Impl-Conv

```
byte Num=33;
int newNum=Num;
```

Expl-Conv

```
double dNum = 9.78;
int iNum = (int) dNum;
Output: 9
```

Problematic

```
int iNum = 130;
byte sByte = (byte) iNum;
Output: -126 (overflow)
```

Simple (Primitive) Data Types	Complex (Reference) Data Types
int, double, char, boolean, etc.	String, Array, Class Objects, etc.
Stores actual values	Stores references (memory addresses)
Fixed amount of memory	Can grow dynamically
Fast and efficient	More flexible but needs more memory

Overflow Value

Byte (if +val)

```
val - 127 = Delta
-128 + (Delta - 1) = OFValue
```

Byte (if -val)

```
val + 128 = |Delta|
128 + (Delta - 1) = OFValue
```

- Wenn eine Zahl außerhalb des gültigen Bereichs liegt, wird sie durch das Zweierkomplement-Verfahren umgewandelt (Wrap-Around-Effekt).
- Negative Werte werden in den positiven Bereich umgewandelt, genauso wie positive Werte in den negativen Bereich übergehen.

Declaration of Variable

- MUST: Start with: Letter, $, _
- CAN'T: Start with a digit
- CAN'T: Use Java reserved words
- CAN: Contain letters, digits, $, _

Common Conv-Errors

- Type Mismatch: int i = "abc";l
- Overflow/Underflow
- Literal Type Suffixes

Conversion-Losses

```
Double: 123.987
To int (truncated): 123

Int: 150
To byte (overflow expected): -106

Negative int: -150
Yo byte (wrap-around): 106
```

Komplexe Datentypen cannot be directly converted into simple data types through assignment
Why: Because they store different kinds of data

Wrapper Classes for PrimDatTyp --> Big Letter

```
int iNum = 42;
// Boxing (Primitive → Object)
Integer oInt = Integer.valueOf(iNum);
// Unboxing (Object → Primitive)
int aInt = oInt.intValue();

// Autoboxing: int → Integer
Integer obj = 5;
// Unboxing: Integer → int
int num = obj;
```

Autoboxing

```
int primitiveInt = 42;
Integer wrappedInteger = primitiveInt;
Output
Primitive int: 42
Autoboxed Integer: 42
Integer wrappedInteger =
Integer.valueOf(primitiveInt);
```

Unboxing

```
Integer wrappedInteger = 100;
int primitiveInt = wrappedInteger;

int primitiveInt = wrappedInteger.intValue();
Output: 100
```

Double to String

```
double originalDouble = 123.456;
String doString = Double.toString(originalDouble);

Printed as "123.436"
```

String to Double

```
double backtoDouble = Double.parseDouble(doString);
```

String text = "Hello";

- The variable text stores the memory address where "Hello" is stored (not the text itself).
- It's like a pointer to the actual data

```
String str = "123";
int num = Integer.parseInt(str); /
System.out.println(num);
```

Simple (Primitive) Data Types	Complex (Reference) Data Types
int, double, char, boolean, etc.	String, Array, Class Objects, etc.
Stores actual values	Stores references (memory addresses)
Fixed amount of memory	Can grow dynamically
Fast and efficient	More flexible but needs more memory

Concept	Example	Automatic?	Risk?
Implicit Type Casting	int → double	✅ Yes	❌ No
Explicit Type Casting	double → int	❌ No	✅ Data loss
Autoboxing	int → Integer	✅ Yes	❌ No
Unboxing	Integer → int	✅ Yes	❌ No
String to Integer	Integer.parseInt("123")	❌ No	✅ NumberFormatException risk
Integer to String	Integer.toString(123)	✅ Yes	❌ No

Why Direct Conversion from Complex Data Types is not allowed

Reason	Explanation
Memory Structure	Primitives store values, while objects store references (memory addresses).
Data Complexity	Objects can store multiple values; primitives hold only one.
No Automatic Extraction	Java doesn't automatically pull numbers from objects or strings.
Different Data Types	A String stores characters, but an int only holds numbers.

Java-CS-P2

Conditional Statements (if | switch)

"Merhfachauswahl"

```
if (a > b) {
   System.out.println("a is greater than b");
   } else if (a < b) {
    System.out.println("b is greater than a");
   } else {
   System.out.println("Both are equal");
}
```

```
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  default:
    System.out.println("Invalid day");
}
```

for Loop

```
for (initialization; condition; update) {
    // Code block
}
```

while Loop

```
while (condition) {
    // Code block
}
```

do-while Loop

```
do {
    // Code block
} while (condition);
```

Loops (for | while | do-while)

Loops repeat a block of code until a condition is met.

for Loop (Definite Loop)

- Used when the number of iterations is known.

while Loop (Indefinite Loop)

- Used when the number of iterations is unknown.
- The loop executes while the condition is true.
- Example: Printing Even Numbers

do-while Loop (Runs at Least Once)

- Executes the block before checking the condition.
- Guarantees at least one execution.
- Example: Taking User Input Until 0 is Entered

```
for (int i = 0; i < 10; i++)
{
System.out.println(i);
}
```

```
int count = 0;
while (count < 5) {
    System.out.println(count);
    count++;
}
```

Type	Example Loops	Condition Known Before Execution?	Use Case
Definite	for loop	✅ Yes (fixed number of iterations)	Counting iterations
Indefinite	while, do-while	❌ No (condition evaluated dynamically)	User input, waiting for a signal

Nested Loops and Control Flow

Loops inside loops are called nested loops.
Used in matrices, patterns, and complex iterations.

break Statement (Exit Loop Early)

Stops execution of the current loop immediately.
Example: Exit When a Condition is Met

```
public class BreakExample {
 public static void main(String[] args) {
     for (int i = 1; i <= 10; i++) {
         if (i == 5) {
           System.out.println("Stopping at " + i);
           break;
         }
         System.out.println(i);
     }
  }
}
```

Output

```
1
2
3
4
Stopping at 5
```

continue Statement (Skip an Iteration)

Skips the current iteration and moves to the next one.
Example: Skip Even Numbers

```
public class ContinueExample {
  public static void main(String[] args) {
      for (int i = 1; i <= 5; i++) {
          if (i % 2 == 0) {
              continue; // Skips even numbers
          }
          System.out.println(i);
      }
  }
}
```

Output:

```
1
3
5
```

Methods

A method in Java is a block of code that performs a specific task.

Method Declaration (Signature)

A method declaration specifies:

- Return type
- Method name
- Parameter list

Method Definition (Implementation)

This includes the method body where the logic is implemented.

Method Invocation (Calling the Method)

Methods are invoked (called) from other parts of the program.
Example: Declaring, Defining, and Calling a Method

```
returnType methodName(parameterType parameterName, ...) {
    // Method body
}
```

```
public class MethodExample {
    // Method Declaration and Definition
    public static void greet(String name) {
        System.out.println("Hello, " + name + "!");
    }

    public static void main(String[] args) {
        // Method Invocation
        greet("Yos");
    }
}
```

Return Types and Parameters

Method with a Return Type

A method with a return type must use return to send back a value.
Example: Method That Returns an Integer

```
public class ReturnExample {
    public static int add(int a, int b) {
        return a + b;  // Returns the sum
    }

    public static void main(String[] args) {
        int sum = add(5, 7);
        System.out.println("Sum: " + sum);
    }
}
```

Method with Multiple Parameters

```
public static double multiply(double x,
double y) {
    return x * y;
}
```

- If a method has a return type, it must include return value
- If a method has void, it does not return anything.

Methodendefinition

- Definiert, was die Methode macht.
- Enthält die Implementierung (Codeblock) der Methode.

Methodenaufruf

- Führt die Methode aus (führt den definierten Code aus).
- Wird von einem anderen Ort im Programm aufgerufen.

Methodendef & Methodenaufruf -- Gemeinsamkeit

- Sowohl die Methodendefinition als auch der Methodenaufruf beziehen sich auf dieselbe Methode und denselben Methodennamen.
- Beide verwenden dieselben Parameter (Typen und Reihenfolge) und Rückgabetypen.

Method Overloading

Method overloading allows multiple methods to have the same name but different parameters.

- Same method name, but different parameters (type or number).
- Java selects the correct method based on arguments.

Java-CS-P3

```java
public class OverloadingExample {
    // Method 1: Add two integers
    public static int add(int a, int b) {
        return a + b;
    }

    // Method 2: Add three integers
    public static int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method 3: Add two double values
    public static double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        System.out.println(add(5, 10));      // Calls method 1
        System.out.println(add(5, 10, 20));  // Calls method 2
        System.out.println(add(5.5, 2.5));   // Calls method 3
    }
}
```

Output
15
35
8.0

Recursion Example

```java
public class RecursionExample {
    public static int factorial(int n) {
        if (n == 0) return 1; // Base case
        return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        System.out.println("Factorial of 5: " + factorial(5));
    }
}
```

Output: Factorial of 5: 120

Implementation of the Euclidean Algorithm

The Euclidean algorithm is used to find the greatest common divisor (GCD).

GCD Recursive Function

```java
public class GCDRecursive {
    public static int gcd(int a, int b) {
        if (b == 0) return a; // Base case
        return gcd(b, a % b);
    }

    public static void main(String[] args) {
        System.out.println("GCD of 48 and 18: " + gcd(48, 18));
    }
}
```

Output: GCD of 48 and 18: 6

GCD Iterative Function

```java
public class GCDIterative {
    public static int gcd(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }

    public static void main(String[] args) {
        System.out.println("GCD of 48 and 18: " + gcd(48, 18));
    }
}
```

Output: GCD of 48 and 18: 6

Recursive vs. Iterative Functions

A recursive function calls itself, whereas an iterative function uses loops.

Recursive Function

A function that calls itself until a base condition is met.
Example: Factorial Using Recursion

Key Features of Recursion:

- Base condition is required to stop infinite recursion.
- Each recursive call creates a new stack frame, which uses more memory.

Iterative Function

- Pros: More efficient
- Cons: Slightly more complex

Recursive Function

- Pros: Easier to understand
- Cons: Uses extra memory (stack calls)

Key Features of Iteration:

- Uses loops, so it is memory efficient.
- Does not use additional stack memory, unlike recursion.

Iterative Function

Uses a loop instead of function calls.
Example: Factorial Using Iteration

```java
public class IterationExample {
    public static int factorial(int n) {
        int result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }

    public static void main(String[] args) {
        System.out.println("Factorial of 5: " + factorial(5));
    }
}
```

Output: Factorial of 5: 120

Concept	Explanation
Method Declaration	Defines method name, return type, and parameters
Method Invocation	Calling the method in main()
Return Type	Specifies what the method returns (void, int, etc.)
Parameters	Pass values into methods (int a, int b)
Method Overloading	Same method name, different parameter list
Recursion	A method calls itself (useful for factorial, GCD, etc.)
Iteration	Uses loops instead of recursive calls
Euclidean Algorithm	Finds GCD using recursion or iteration

Array

An array in Java is an indexed collection of elements, where:

- Index starts from 0.
- The size is fixed after declaration.
- Arrays must be declared and initialized before use.

```java
int[] numbers = new int[5];
// Declaration with size 5
numbers[0] = 10; // Assigning values
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;
```

```java
int[] numbers = {10, 20, 30, 40, 50};
// Direct initialization
```

2D Arrays (Matrix)

A 2D array is an array of arrays (matrix-like structure).

```java
int[][] matrix = new int[3][3];
// 3x3 matrix

int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

Traversing 1D Array using for and foreach

For

```java
public class ArrayTraversal {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};

        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Index " + i + ": " + numbers[i]);
        }
    }
}
```

Output:
Index 0: 10
Index 1: 20
Index 2: 30
Index 3: 40
Index 4: 50

Foreach

```java
public class ForeachExample {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};

        for (int num : numbers) {
            // Reads elements directly
            System.out.println(num);
        }
    }
}
```

Output:
10
20
30
40
50

Traversing a 2D Array Using Nested Loops

```java
public class TwoDArrayTraversal {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        for (int i = 0; i < matrix.length; i++) { // Rows
            for (int j = 0; j < matrix[i].length; j++) {
                // Columns
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println(); // New line after each row
        }
    }
}
```

Output:
1 2 3
4 5 6
7 8 9

- matrix.length → Number of rows.
- matrix[i].length → Number of columns (in row i).

Java-CS-P4

Traversing 2D Arrays Using foreach

```
public class TwoDArrayForeach {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        for (int[] row : matrix) {
            for (int num : row) {
                System.out.print(num + " ");
            }
            System.out.println();
        }
    }
```

Loop Type	Usage	Best for
for loop	Uses index (numbers[i])	When index is needed
foreach loop	Directly accesses elements	When index is not needed

---

Matrix Operations

```
public class MatrixAddition {
    public static void main(String[] args) {
      int[][] A = {
          {1, 2, 3},
          {4, 5, 6}
      };
      int[][] B = {
          {7, 8, 9},
          {10, 11, 12}
      };
      int[][] result = new int[2][3];

      for (int i = 0; i < A.length; i++) { // Rows
          for (int j = 0; j < A[i].length; j++) { // Columns
              result[i][j] = A[i][j] + B[i][j];
          }
      }

      // Print the result matrix
      for (int[] row : result) {
          for (int num : row) {
              System.out.print(num + " ");
          }
          System.out.println();
      }
    }
}
```

Output:

```
8 10 12
14 16 18
```

Concept	Example	Key Points
1D Array Declaration	int[] arr = new int[5];	Stores a sequence of elements
1D Array Initialization	int[] arr = {1,2,3};	Initializes values directly
1D Array Traversal	for (int i = 0; i < arr.length; i++)	Uses for or foreach loops
2D Array Declaration	int[][] matrix = new int[3][3];	Rows and columns format
2D Array Initialization	int[][] matrix = {{1,2}, {3,4}};	Shorthand initialization
2D Array Traversal	Nested loops (for or foreach)	Access row-wise elements
Matrix Addition	result[i][j] = A[i][j] + B[i][j];	Adds two matrices
Matrix Multiplication	Three nested loops	Uses sum-product rule

Matrix Multiplications

```
public class MatrixMultiplication {
  public static void main(String[] args) {
      int[][] A = {
          {1, 2},
          {3, 4}
      };
      int[][] B = {
          {5, 6},
          {7, 8}
      };
      int rows = A.length;
      int cols = B[0].length;
      int commonDim = B.length;

      int[][] result = new int[rows][cols];

      for (int i = 0; i < rows; i++) {
          for (int j = 0; j < cols; j++) {
              for (int k = 0; k < commonDim; k++) {
                  result[i][j] += A[i][k] * B[k][j];
              }
          }
      }

      // Print result matrix
      for (int[] row : result) {
          for (int num : row) {
              System.out.print(num + " ");
          }
          System.out.println();
      }
  }
}
```

Output:

19 22
43 50

- Outer loop: Iterates over rows of A.
- Middle loop: Iterates over columns of B.
- Inner loop: Multiplies elements and sums them up.

Java-CS-P5

Object Oriented Programming

It models real-world entities using classes and objects.

Encapsulation, Abstraction, Inheritance, and Polymorphism.

A class is a blueprint for objects. It defines properties (fields/variables) and behaviors (methods).

An object is an instance of a class with unique values.

```
// Defining a class
public class Car {
    String brand;
    int speed;

    // Method to display car details
    void display() {
        System.out.println("Car brand: " + brand + ", Speed: " +
speed + " km/h");
    }
}

// Creating an object in main
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); // Object instantiation
        myCar.brand = "Toyota";
        myCar.speed = 120;
        myCar.display(); // Call method
    }
}
```

Output:

```
Car brand: Toyota, Speed: 120 km/h
```

Key Points:

- **Car is the class.**
- **myCar is an object.**
- **Objects store data (brand, speed).**

Methods (display()) define behavior.

Constructor and Instantiation

A constructor is a special method used to initialize objects.

Constructor Rules

- Same name as class.
- No return type.
- Automatically called when an object is created.

Example: Constructor Usage

```
public class Student {
    String name;
    int id;

    // Constructor
    public Student(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public void display() {
        System.out.println("Student Name: " + name + ", ID: " + id);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student1 = new Student("John", 101);
        student1.display();
    }
}
```

Output:

Student Name: John, ID: 101

- The constructor initializes name and id when new Student(...) is called.
- No need for manual assignment (this.name = ...).

Encapsulation and Access Modifiers

Encapsulation means hiding the internal state of an object and allowing controlled access via getters and setters.

Access Modifiers

Modifier	Scope
private	Only within the same class
public	Accessible from anywhere
protected	Accessible in the same package and subclasses
(default)	Accessible within the same package

Example: Encapsulation Using private Variables and Getters/Setters

```
public class Person {
    private String name; // Private variable
    private int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter method
    public String getName() {
        return name;
    }

    // Setter method
    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age > 0) { // Basic validation
            this.age = age;
        }
    }

    public void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

// Using encapsulation in main
public class Main {
    public static void main(String[] args) {
        Person person = new Person("Alice", 25);
        person.display();

        person.setAge(30);
        System.out.println("Updated Age: " + person.getAge());
    }
}
```

Output:

Name: Alice, Age: 25

Updated Age: 30

Encapsulation Benefits:

- Data hiding: private prevents direct access.
- Controlled access: getters and setters allow validation.

Java-CS-P6

toString() Method

The toString() method converts an object into a readable String.

Example: Using toString()

```
public class Book {
    String title;
    double price;

    public Book(String title, double price) {
        this.title = title;
        this.price = price;
    }

    // Overriding toString() method
    @Override
    public String toString() {
        return "Book: " + title + ", Price: $" + price;
    }
}

public class Main {
    public static void main(String[] args) {
        Book book = new Book("Java Basics", 29.99);
        System.out.println(book.toString()); // Implicitly called
    }
}
```

Output:
Book: Java Basics, Price: $29.99

Why Use toString()?

Improves readability when printing objects.

Concept	Description	Example
Class & Object	Blueprint for creating objects	`class Car { ... }`
Encapsulation	Data hiding using private	`private String name;`
Access Modifiers	public, private, protected	`public void setName()`
Constructor	Initializes object	`public Car(String brand) { … }`
toString() Method	Converts object to string	`System.out.println(object);`
UML-based Class	Class diagram to Java code	`+ getSalary(): double`

Implementing UML-Based Classes

UML (Unified Modeling Language) represents class relationships using diagrams.

Key UML Concepts:

- Attributes (name, salary) are private.
- Methods (getName(), getSalary()) provide controlled access.

Java Code for UML Class

```
public class Employee {
    private String name;
    private double salary;

    // Constructor
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    // Getter methods
    public String getName() {
        return name;
    }

    public double getSalary() {
        return salary;
    }

    public void display() {
        System.out.println("Employee Name: " + name + ", Salary: $" + salary);
    }
}

// Using Employee class
public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee("Alice", 50000);
        emp.display();
    }
}
```

Output:
Employee Name: Alice, Salary: $50000.0

UML Example

```
Class: Employee
+--------------------+
| Employee |
+--------------------+
| - name: String |
| - salary: double |
+--------------------+
| + Employee(name, salary) |
| + getName(): String |
| + getSalary(): double |
+--------------------+
```

Debugging

Errors in Java

- Syntax Errors (Compile-time errors)
- Logical Errors (Run-time errors)

Common Syntax Errors

Error Type	Cause	Example Code (Wrong)	Fixed Code (Correct)
Missing Semicolon (;)	Every statement must end with ;	int x = 10	int x = 10;
Misspelled Keyword	Java keywords must be correct	`public static void mian(String[] args) {}`	`public static void main(String[] args) {}`
Incorrect Variable Declaration	Java is strongly typed	int number = "Hello";	`String number = "Hello";`
Mismatched Brackets {}	All { must have }	`if (x > 10) { System.out.println(x);`	`if (x > 10) { System.out.println(x); }`
Incompatible Data Types	Cannot assign wrong type	`int num = 5.5;`	`double num = 5.5;`
Using Undeclared Variables	Variables must be declared before use	`System.out.println(count);`	`int count = 0; System.out.println(count);`

Common Logical Errors

Error Type	Cause	Example Code (Wrong)	Fixed Code (Correct)
Wrong Condition in if Statement	Condition does not match logic	if (x = 10) {} (= instead of ==)	if (x == 10) {}
Wrong Loop Condition	Infinite loops or incorrect exit	`while (x != 10) { x--; }` (x never reaches 10)	`while (x > 10) { x--; }`
Incorrect Order of Operations	Wrong mathematical calculations	int result = 10 / 2 * 5; (Does not give expected result)	`int result = (10 / 2) * 5;`
Variable Scope Issue	Using variables outside the declared scope	`System.out.println(num);` (Declared inside block)	Declare num outside the block
Incompatible Data Types	Cannot assign wrong type	`int num = 5.5;`	`double num = 5.5;`
Using == Instead of .equals() for Strings	Strings must be compared using .equals()	`if (name == "John")`	`if (name.equals("John"))`

Java-CS-P7

Debugging Techniques

Debugging is the process of finding and fixing errors in a program.

Using Print Statements (System.out.println())

- Best for quick debugging.
- Print variable values at different points in the code.

```
Example: Debugging with System.out.println()
public class DebugExample {
    public static void main(String[] args) {
        int num = 5;
        System.out.println("Before loop: " + num);

        while (num > 0) {
            System.out.println("Current value: " + num);
            num--; // Reduce num each iteration
        }

        System.out.println("After loop: " + num);
    }
}
```

Handling Exceptions (try-catch)

Exception handling prevents the program from crashing.

```
Example: Catching NumberFormatException
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            String str = "abc";
            int num = Integer.parseInt(str); // ❌ Error: Not a number
        } catch (NumberFormatException e) {
            System.out.println("Error: Invalid number format!");
        }
    }
}
```

Output:
Error: Invalid number format!

Using a Debugger in an IDE (Eclipse, IntelliJ)

- Set Breakpoints: Stop execution at a specific line.
- Step Through Code: Run one line at a time.
- Inspect Variables: Check values at each step.

Concept	Description	Fixing Method
Syntax Errors	Code does not compile	Fix spelling, missing ;, incorrect brackets {}
Logical Errors	Code runs but gives wrong output	Check conditions, calculations, and loops
Print Debugging	Printing variables to check values	Use System.out.println()
Using Debugger	Step through code	Set breakpoints in IDE (Eclipse, IntelliJ)
Handling Exceptions	Prevents program crash	Use try-catch for error handling