Object-Oriented Programming
Java Fundamentals

Java Language Essentials for OOP

Conditional logic, loops, methods, and exception handling — with reusable patterns and small examples.

How to use this handout

This is a compact reference for everyday Java inside classes and methods. Read each section once, then keep it nearby while you code.

A minimal Java program

Most course projects live inside a class. The JVM begins execution at a method named main.

```
public class App {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
```

Java is statically typed: each variable and method has a declared type. Values are either primitives (like int, double, boolean) or references to objects (like String and your own classes).

Conditional logic

Conditionals choose a path based on a boolean expression. Common operators include ==, !=, <, <=, >, >=, and connectors &&, ||, !.

if / else-if / else

```
int score = 83;

if (score >= 90) {
    System.out.println("A");
} else if (score >= 80) {
    System.out.println("B");
} else if (score >= 70) {
    System.out.println("C");
} else {
    System.out.println("Needs improvement");
}
```

Tip: use braces while learning. Compare String values with equals, not ==.

switch (discrete options)

```
String day = "MON";
```

```
switch (day) {
    case "MON":
    case "TUE":
    case "WED":
    case "THU":
    case "FRI":
        System.out.println("Weekday");
        break;
    case "SAT":
    case "SUN":
        System.out.println("Weekend");
        break;
    default:
        System.out.println("Unknown");
}
```

Tip: in classic switch, do not forget break unless you intentionally want fall-through.

Loops

Loops repeat work while a condition holds or while elements remain. Use for when you know the count; use while when you stop based on a condition discovered during execution.

for (index-based)

```
int sum = 0;
for (int i = 0; i < numbers.length; i++) {
    sum += numbers[i];
}
```

enhanced for (each element)

```
int sum = 0;
for (int n : numbers) {
    sum += n;
}
```

Use the enhanced for loop when you do not need the index. Inside loops, break exits early; continue skips to the next iteration (use sparingly).

while / do-while (condition-driven)

```
Scanner scanner = new Scanner(System.in);

int value;
do {
    System.out.print("Enter a positive number: ");
    value = Integer.parseInt(scanner.nextLine());
} while (value <= 0);
```

A do-while loop runs at least once, which is useful for prompts.

Methods

A method is a named block of code that can take inputs (parameters) and produce an output (a return value). Methods express behavior and reduce duplication.

Signature, parameters, return values

```
public class MathUtils {

    // access-modifier static return-type name(parameters)
    public static int clamp(int value, int min, int max) {
        if (value < min) return min;
        if (value > max) return max;
        return value;
    }
}
```

Use void when a method performs an action but does not return a value. Java passes arguments by value (including object references).

Instance vs. static

```
public class BankAccount {
    private double balance;

    public BankAccount(double startingBalance) {
        this.balance = startingBalance;
    }

    public void deposit(double amount) {
        if (amount <= 0) throw new IllegalArgumentException("amount must be
positive");
        balance += amount;
    }

    public double getBalance() {
        return balance;
    }
}
```

Instance methods operate on a particular object's fields (like balance). Static methods belong to the class and are often used for utilities or factories.

Overloading (same name, different parameters)

```
public static int max(int a, int b) { return (a > b) ? a : b; }
public static double max(double a, double b) { return (a > b) ? a : b; }
```

Overloading reuses a name for related operations with different parameter types or counts.

Object-Oriented Programming

Java Fundamentals

Exception handling

Exceptions represent unexpected or invalid situations. Use them to separate normal logic from error-handling logic.

try / catch / finally

```
try {
    int n = Integer.parseInt(text);
    System.out.println("Parsed: " + n);
} catch (NumberFormatException ex) {
    System.out.println("Not an integer: " + text);
} finally {
    System.out.println("Done.");
}
```

Catch the most specific exception you can reasonably handle. Use finally for cleanup that must happen regardless of success.

Throwing exceptions (fail fast on bad input)

```
public void setAge(int age) {
    if (age < 0) throw new IllegalArgumentException("age must be non-
negative");
    this.age = age;
}
```

try-with-resources (preferred cleanup for I/O)

```
try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
    System.out.println(br.readLine());
} catch (IOException ex) {
    System.out.println("Could not read file: " + ex.getMessage());
}
```

Any object that implements AutoCloseable can be used in try-with-resources and will be closed automatically.

Quick checklist

- Compare Strings with equals and keep conditions readable.
- Choose the loop that makes the stopping rule obvious.
- Write small, well-named methods; prefer instance methods for behavior and static methods for utilities.
- Catch specific exceptions; use try-with-resources for I/O.