

INTRODUCTION TO PROGRAMMING IN JAVA

Marko Šutić, Marija Jurinec

WIFI

- Business club 5 Guest
- password: Bguest5!

PREREQUISITES

- IntelliJ community edition
- Exercises: <https://github.com/mithril-eu/course-java-beginner-exercises>

SETUP

- JAVA_HOME environment variable
 - java -version
- HelloWorld.java

DAY 1: INTRODUCTION TO JAVA AND BASIC PROGRAMMING CONCEPTS

Learning Objectives

- Understand what Java is and its role in software development
- Set up Java development environment (JDK and IDE)
- Write and execute basic Java programs
- Learn about variables, data types, and basic operators
- Understand basic program structure and syntax

INTRODUCING JAVA

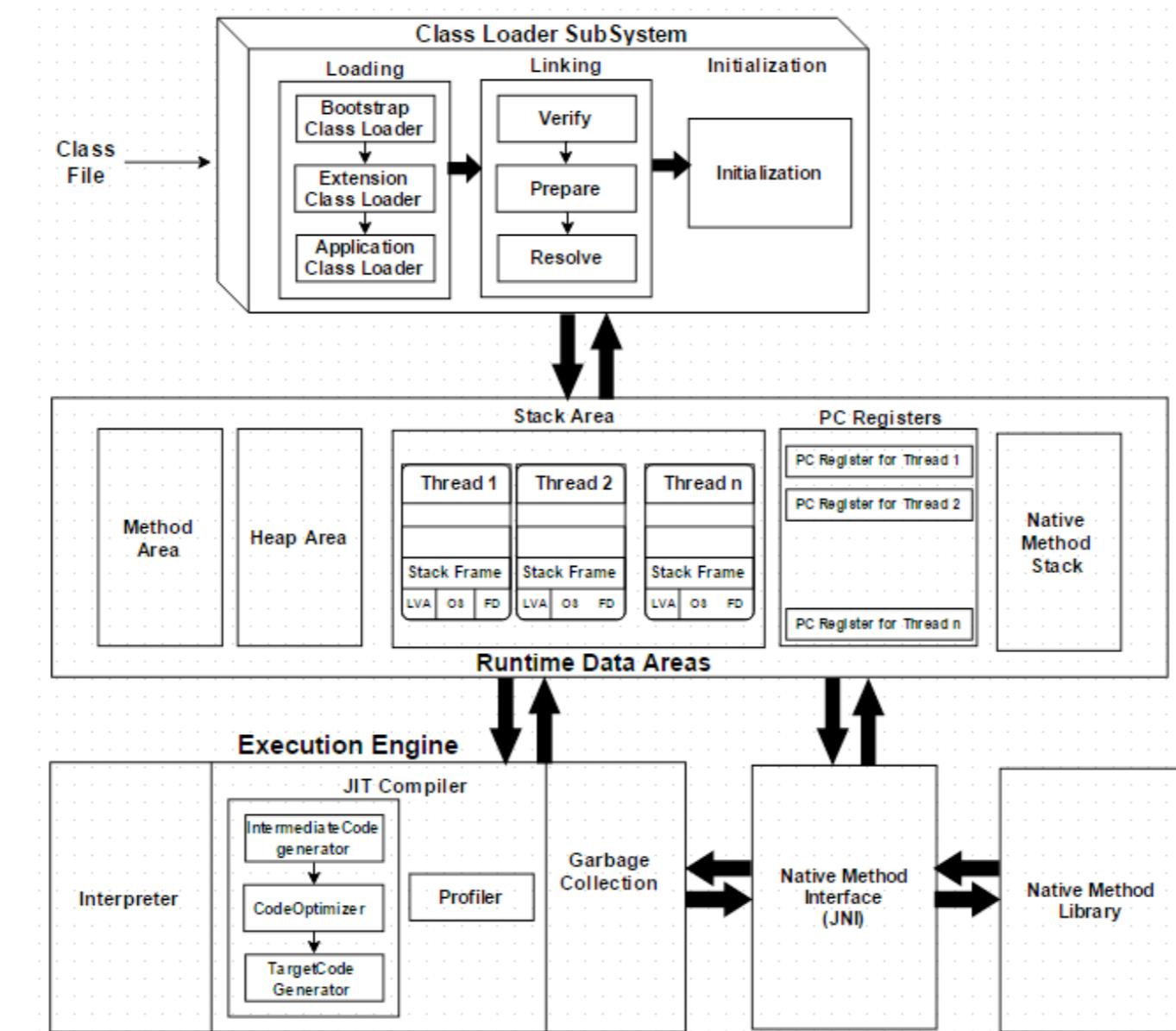
- Created by James Gosling at Sun Microsystems (1991-1995)
- Revolutionary "Write Once, Run Anywhere" approach
- Core principles: Object-oriented, Simple, Secure, Portable
- Gained popularity through internet applications (Applets)
- Acquired by Oracle Corporation in 2010
- #1 language for enterprise applications
- Official language for Android development
- Extensive open-source ecosystem
- Consistently among top 3 programming languages

JAVA ENVIRONMENT

THE JAVA ECOSYSTEM

- Java source code (.java) → Java bytecode (.class) → JVM execution
- Write once, run anywhere (WORA) principle
- Multi-layer platform architecture:
 - Java Development Kit (JDK)
 - Java Runtime Environment (JRE)
 - Java Virtual Machine (JVM)
- Platform independence through bytecode

JVM ARCHITECTURE



HELLO WORLD IN JAVA

```
1 package eu.mithril.java.edu1;
2 /**
3  * Javadoc comment
4 */
5 public class HelloWorld {
6     public static void main(String[] args) {
7         System.out.println("Hello, World!");
8         // System.out.println("This line is not executed");
9         /*
10         System.out.println("This line is ignored by compiler");
11         System.out.println("Including this");
12         */
13     }
14 }
```

TYPES AND VARIABLES

DECLARING AND INITIALIZING

```
1 DataType variableName = initialization;
```

- The case in the name of a variable makes a difference.
- Variables can store not only strings and numbers, but also characters and other data types which we will learn about later in the next topics.

```
String language = "java";
int numberOfApples = 5;
```

ACCESSING VARIABLES

```
1 String dayOfWeek = "Monday";
2 System.out.println(dayOfWeek); /* Monday */
3
4 /* Assigning one variable to another */
5 int one = 1;
6 int num = one;
7 System.out.println(num); /* 1 */
```

MODIFYING VARIABLES

```
1 String dayOfWeek = "Monday";
2 System.out.println(dayOfWeek); /* Monday */
3
4 dayOfWeek = "Tuesday";
5 System.out.println(dayOfWeek); /* Tuesday */
6
7 /* Type checking example */
8 int number = 10;
9 number = 11;           /* ✓ OK */
10 number = "twelve";   /* ✗ Wrong! Type mismatch */
```

ALTERNATIVE DECLARATION FORMS

```
1 /* Multiple variables in one line */
2 String language = "java", version = "8 or newer";
3
4 /* Separate declaration and initialization */
5 int age;      /* declaration */
6 age = 35;     /* initialization */
```

TYPE INFERENCE (JAVA 10+)

```
1 var language = "Java";      /* Compiler infers String */
2 var version = 10;           /* Compiler infers int */
```

DATA TYPES AND THEIR SIZES

INTEGER TYPES

```
1 /* byte: 8 bits (-128 to 127) */
2 byte smallNumber = 100;
3
4 /* short: 16 bits (-32,768 to 32,767) */
5 short mediumNumber = 32000;
6
7 /* int: 32 bits (-2^31 to 2^31-1) */
8 int standardNumber = 2147483647;
9
10 /* long: 64 bits (-2^63 to 2^63-1) */
11 long bigNumber = 9223372036854775807L;
12
13 /* Using underscore for readability */
14 long million = 1_000_000L;
```

FLOATING-POINT AND CHARACTER TYPES

```
1 /* double: 64 bits (~14-16 decimal digits) */
2 double pi = 3.14159265359;
3 double scientificNotation = 1.23e-4;
4
5 /* float: 32 bits (~6-7 decimal digits) */
6 float price = 19.99f; /* f suffix required */
7
8 /* char: 16 bits (Unicode characters) */
9 char letter = 'A';
10 char symbol = '$';
11 char unicode = '\u0041'; /* Unicode for 'A' */
```

BOOLEAN TYPE AND TYPE PROPERTIES

```
1 /* boolean: true or false */
2 boolean isActive = true;
3 boolean hasError = false;
4
5 /* Type size demonstration */
6 System.out.println("Size in bits:");
7 System.out.println("byte: " + Byte.SIZE);      /* 8 bits */
8 System.out.println("short: " + Short.SIZE);    /* 16 bits */
9 System.out.println("int: " + Integer.SIZE);     /* 32 bits */
10 System.out.println("long: " + Long.SIZE);       /* 64 bits */
11 System.out.println("float: " + Float.SIZE);     /* 32 bits */
12 System.out.println("double: " + Double.SIZE);   /* 64 bits */
13 System.out.println("char: " + Character.SIZE);  /* 16 bits */
```

PRIMITIVE AND REFERENCE TYPES

DATA TYPES OVERVIEW

- **Primitive Types:** Built-in, lowercase (e.g., `int`)
- **Reference Types:** Extensible, uppercase (e.g., `String`)

CREATING REFERENCE TYPES

```
1 /* Using new keyword */
2 String language = new String("java");
3
4 /* Using string literal (simpler) */
5 String language = "java";
```

MEMORY: STACK VS HEAP

```
1 /* Primitive: value is copied */
2 int a = 100;
3 int b = a;      /* b gets copy of 100 */
4
5 /* Reference: address is copied */
6 String s1 = new String("java");
7 String s2 = s1;      /* s2 points to same object */
```

COMPARING REFERENCE TYPES

```
1 String s1 = new String("java");
2 String s2 = new String("java");
3 String s3 = s2;
4
5 /* Compare references */
6 System.out.println(s1 == s2);      /* false */
7 System.out.println(s2 == s3);      /* true */
8
9 /* Compare content */
10 System.out.println(s1.equals(s2)); /* true */
11 System.out.println(s2.equals(s3)); /* true */
```

THE NULL VALUE

```
1 /* Reference types can be null */
2 String str = null;
3 System.out.println(str); /* prints: null */
4 str = "hello";
5 System.out.println(str); /* prints: hello */
6
7 /* Primitive types cannot be null */
8 int n = null; /* Won't compile! */
```

CONSTANTS. FINAL VARIABLES

WHAT ARE CONSTANTS?

```
1 final double PI = 3.1415;
2 final String HELLO_MSG = "Hello";
3
4 System.out.println(PI);      /* 3.1415 */
5 System.out.println(HELLO_MSG); /* Hello */
```

NAMING CONVENTIONS AND RULES

```
1 /* Won't compile - can't modify final variable */
2 final double PI = 3.1415;
3 PI = 3.1416; /* Compilation error!
4
5 /* Must initialize before use */
6 final boolean FALSE;
7 System.out.println(FALSE); /* Error: might not be initialized */
8
9 /* Correct initialization */
10 final boolean FALSE;
11 FALSE = false;
12 System.out.println(FALSE); /* OK */
```

FINAL REFERENCE VARIABLES

```
1 /* Cannot reassign reference */
2 final StringBuilder builder = new StringBuilder();
3 builder = new StringBuilder(); /* Error! */
4
5 /* Can modify object state */
6 final StringBuilder sb = new StringBuilder();
7 sb.append("Hello!"); /* OK */
8 System.out.println(sb.toString()); /* Hello! */
```

TYPE INFERENCE WITH FINAL

```
1 /* Type inference with constants */
2 final var FINAL_VAR = 10;          /* inferred as int */
3 final var MSG = "Hello!";          /* inferred as String */
4
5 /* Regular variables can still use final values */
6 final int count = 10;
7 int cnt = count;
8 cnt = 20; /* OK - cnt is not final */
```

ENUMS

WHAT ARE ENUMS?

```
1 /* Basic enum declaration */
2 public enum Season {
3     SPRING, SUMMER, AUTUMN, WINTER
4 }
5
6 /* Using an enum */
7 Season current = Season.SUMMER;
```

DEFINING ENUMS

```
1 /* Enum inside a class */
2 class Weather {
3     enum Condition {
4         SUNNY, RAINY, CLOUDY, SNOWY
5     }
6 }
7
8 /* Standalone enum */
9 public enum UserStatus {
10    PENDING, ACTIVE, BLOCKED
11 }
```

WORKING WITH ENUMS

```
1 /* Creating enum variables */
2 UserStatus status = UserStatus.ACTIVE;
3
4 /* Getting enum name */
5 String statusName = status.name(); /* ACTIVE */
6
7 /* Converting String to enum */
8 UserStatus blocked = UserStatus.valueOf("BLOCKED");
9
10 /* Using in control flow */
11 if (status == UserStatus.ACTIVE) {
12     /* Process active user */
13 }
```

TYPE CASTING

IMPLICIT CASTING (WIDENING)

```
1 /* Implicit casting happens automatically */
2 int num = 100;
3 long bigNum = num;          /* int to long */
4
5 short shortNum = 100;
6 int regular = shortNum;    /* short to int */
7
8 char ch = 'A';
9 int code = ch;             /* char to int (gets ASCII/Unicode value) */
10
11 long bigVal = 1000L;
12 double decimal = bigVal;  /* long to double */
```

EXPLICIT CASTING (NARROWING)

```
1 /* Explicit casting requires parentheses */
2 double price = 23.5;
3 int whole = (int) price;      /* loses decimal portion */
4
5 long bigNum = 1234567L;
6 int smaller = (int) bigNum; /* possible data loss */
7
8 /* Multiple casts in expression */
9 double result = 3.75;
10 short small = (short)(int) result;
11
12 /* Char casting */
13 int ascii = 65;
14 char letter = (char) ascii; /* becomes 'A' */
```

CASTING PITFALLS AND BEST PRACTICES

```
1 /* Overflow example */
2 long bigValue = 100_000_000_000L;
3 int truncated = (int) bigValue; /* causes overflow */
4
5 /* Precision loss example */
6 long preciseNum = 1_200_000_002L;
7 float imprecise = preciseNum; /* becomes 1.2E9 */
8
9 /* Redundant casting - avoid this */
10 int small = 100;
11 long bigger = (long) small; /* unnecessary cast */
12
13 /* Invalid casting with boolean */
14 boolean flag = true;
15 /* int number = (int) flag; */ /* Won't compile! */
```

OPERATIONS ON PRIMITIVE TYPES

BINARY ARITHMETIC OPERATORS

```
1 /* Basic arithmetic operations */
2 System.out.println(13 + 25); /* 38 (addition) */
3 System.out.println(70 - 30); /* 40 (subtraction) */
4 System.out.println(21 * 3); /* 63 (multiplication) */
5 System.out.println(8 / 3); /* 2 (integer division) */
6 System.out.println(10 % 3); /* 1 (remainder) */
7
8 /* Complex expression */
9 System.out.println(1 + 3 * 4 - 2); /* 11 */
10 System.out.println((1 + 3) * (4 - 2)); /* 8 */
```

INCREMENT AND DECREMENT

```
1 /* Prefix increment */
2 int a = 4;
3 int b = ++a; /* a is incremented first */
4 System.out.println(a); /* 5 */
5 System.out.println(b); /* 5 */
6
7 /* Postfix increment */
8 int x = 4;
9 int y = x++; /* x's original value is used first */
10 System.out.println(x); /* 5 */
11 System.out.println(y); /* 4 */
```

RELATIONAL OPERATORS

```
1 int one = 1;
2 int two = 2;
3
4 /* Basic comparisons */
5 boolean isEqual = one == one;      /* true */
6 boolean notEqual = one != two;     /* true */
7 boolean lessThan = one < two;       /* true */
8 boolean greaterEqual = two >= one; /* true */
9
10 /* With arithmetic */
11 boolean result = one + 10 > two + 5; /* true */
12
13 /* Range check */
14 boolean inRange = number > 100 && number < 200;
```

OPERATOR PRECEDENCE

Highest to lowest:

1. Parentheses ()
2. Unary ` , ` - ` , ` + , --
3. Multiplication, Division, Remainder * , / , %
4. Addition, Subtraction + , -
5. Relational operators < , ≤ , > , ≥
6. Equality operators == , !=

```
1 /* Example combining different operators */
2 int x = 5;
3 boolean result = (x++ * 3) > (--x + 5);
```

CONTROL FLOW STATEMENTS

CONDITIONAL STATEMENT (IF-ELSE)

```
1 /* Basic if-else */
2 if (age > 100) {
3     System.out.println("Very experienced person");
4 }
5
6 /* Multiple conditions */
7 if (dollars < 1000) {
8     System.out.println("Buy a laptop");
9 } else if (dollars < 2000) {
10    System.out.println("Buy a personal computer");
11 } else {
12    System.out.println("Buy a server");
13 }
```

TERNARY OPERATOR

```
1 /* Finding maximum */
2 int max = a > b ? a : b;
3
4 /* Even or odd check */
5 String result = num % 2 == 0 ? "even" : "odd";
6
7 /* Nested ternary (use carefully) */
8 String compare = a == b ? "equal" :
9                 a > b ? "more" : "less";
```

WHY SWITCH?

```
1 /* Multiple if-else can be hard to read */
2 if (action == 1) {
3     System.out.println("Start game");
4 } else if (action == 2) {
5     System.out.println("Load game");
6 } else if (action == 3) {
7     System.out.println("Help");
8 }
9
10 /* Cleaner with switch */
11 switch (action) {
12     case 1:
13         System.out.println("Start game");
14         break;
15     case 2:
16         System.out.println("Load game");
17         break;
18     case 3:
19         System.out.println("Help");
20         break;
21 }
```

SWITCH SYNTAX

```
1 switch (variable) {  
2     case value1:  
3         /* code for value1 */  
4         break;  
5     case value2:  
6         /* code for value2 */  
7         break;  
8     default:  
9         /* code for all other values */  
10        break; /* optional in default */  
11 }
```

COMMON PITFALLS

```
1 int val = 1;
2 switch (val) {
3     case 0:
4         System.out.println("zero");
5         break;
6     case 1:
7         System.out.println("one");
8         /* Missing break - will fall through! */
9     case 2:
10        System.out.println("two");
11        break;
12    default:
13        System.out.println("other value");
14 }
15 /* Prints: one two */
```

FOR LOOP

```
1 /* Basic for loop */
2 for (int i = 0; i < 10; i++) {
3     System.out.print(i + " ");
4 }
5
6 /* Nested loops */
7 for (int i = 1; i < 10; i++) {
8     for (int j = 1; j < 10; j++) {
9         System.out.print(i * j + "\t");
10    }
11    System.out.println();
12 }
```

FOR-EACH LOOP

```
1 /* Basic array iteration example */
2 int[] numbers = { 1, 2, 3, 4, 5 };
3
4 /* Using for-each loop */
5 for (int number : numbers) {
6     System.out.print(number + " ");
7 }
8 /* Output: 1 2 3 4 5 */
```

FOR-EACH SYNTAX

```
1 /* Syntax demonstration */
2 String[ ] fruits = {"Apple", "Banana", "Orange"};
3
4 /* Each element accessed directly */
5 for (String fruit : fruits) {
6     System.out.println(fruit);
7 }
8
9 /* Compared to traditional for loop */
10 for (int i = 0; i < fruits.length; i++) {
11     System.out.println(fruits[i]);
12 }
```

WHEN TO USE FOR-EACH

```
1 /* Good use case: simple processing */
2 int[] scores = {85, 92, 78, 95, 88};
3 int sum = 0;
4
5 for (int score : scores) {
6     sum += score;
7 }
8
9 /* Bad use case: need index */
10 /* Cannot use for-each when you need array position */
11 for (int i = 0; i < scores.length; i++) {
12     System.out.println("Score " + (i + 1) + ": " + scores[i]);
13 }
```

WHILE AND DO-WHILE LOOPS

```
1 /* Basic while loop structure */
2 while (condition) {
3     /* loop body */
4 }
5
6 /* Example: counting up to 5 */
7 int i = 0;
8 while (i < 5) {
9     System.out.println(i);
10    i++;
11 }
12
13 /* Example: printing alphabet */
14 char letter = 'A';
15 while (letter <= 'Z') {
16     System.out.print(letter);
17     letter++;
18 }
```

DO-WHILE LOOP

```
1 /* Basic do-while structure */
2 do {
3     /* loop body */
4 } while (condition);
5
6 /* Example: reading input until zero */
7 int value;
8 do {
9     value = scanner.nextInt();
10    System.out.println(value);
11 } while (value != 0);
```

COMPARING WHILE VS DO-WHILE

```
1 /* While loop might never execute */
2 int x = 10;
3 while (x < 10) {
4     System.out.println("Never prints");
5 }
6
7 /* Do-while always executes once */
8 int y = 10;
9 do {
10     System.out.println("Prints once");
11 } while (y < 10);
12
13 /* Infinite loop example */
14 while (true) {
15     /* runs forever unless broken */
16 }
```

BREAK AND CONTINUE

```
1 /* break example */
2 while (true) {
3     if (condition) {
4         break; /* exits loop
5     }
6 }
7
8 /* continue example */
9 for (int i = 0; i < n; i++) {
10    if (i % 2 != 0) {
11        continue; /* skips odd numbers */
12    }
13    System.out.print(i + " ");
14 }
```

STRING

WHAT IS STRING?

```
1 /* Creating strings */
2 String message = "Hello, Java";          /* literal */
3 String str = new String("my-string");     /* using new */
4 String empty = "";                      /* empty string */
5 String nullStr = null;                  /* null string */

6
7 /* Basic operations */
8 int length = message.length();          /* 10 */
9 char firstChar = message.charAt(0);      /* 'H' */
10 char lastChar = message.charAt(length-1); /* 'a' */
```

COMMON STRING METHODS

```
1 String text = "The Simple Text";
2
3 /* Case conversion */
4 String upper = text.toUpperCase();      /* "THE SIMPLE TEXT" */
5 String lower = text.toLowerCase();      /* "the simple text" */
6
7 /* Checking content */
8 boolean isEmpty = text.isEmpty();       /* false */
9 boolean starts = text.startsWith("The"); /* true */
10 boolean contains = text.contains("Simple"); /* true */
11
12 /* Modifying content */
13 String noSpaces = text.replace(" ", ""); /* "TheSimpleText" */
14 String trimmed = text.trim();           /* removes whitespace */
15 String sub = text.substring(4, 10);     /* "Simple" */
```

STRING CONCATENATION

```
1 /* Basic concatenation */
2 String first = "John";
3 String last = "Smith";
4 String full = first + " " + last;      /* "John Smith" */
5
6 /* Using concat method */
7 String fullName = first.concat(" ").concat(last);
8
9 /* Mixed type concatenation */
10 String result = "Count: " + 42;        /* "Count: 42" */
11 String mixed = "Value: " + 10 + true; /* "Value: true" */
12
13 /* Order matters */
14 String str = "str";
15 int num = 100;
16 String res1 = str + num + 50; /* "str10050" */
17 String res2 = num + 50 + str; /* "150str" */
```

COMPARING STRINGS

```
1 String s1 = "hello";
2 String s2 = "HELLO";
3 String s3 = "hello";
4
5 /* Correct string comparison */
6 boolean equals1 = s1.equals(s3);                      /* true */
7 boolean equals2 = s1.equals(s2);                      /* false */
8 boolean equalsIgnoreCase = s1.equalsIgnoreCase(s2); /* true */
9
10 /* Wrong comparison! */
11 boolean wrong = (s1 == s3);                         /* don't do this! */
```

FORMATTED OUTPUT

```
1 /* Basic formatting */
2 System.out.printf("My name is %s. Age: %d", "John", 25);
3
4 /* String.format */
5 String formatted = String.format("Hello, %s!", "World");
6 System.out.println(formatted);
```

FORMAT SPECIFIERS

```
1 /* Integer formatting */
2 System.out.printf("Number: %d%n", 15000);
3
4 /* Floating point with precision */
5 System.out.printf("Price: %.2f%n", 15.23);
6
7 /* Multiple values */
8 System.out.printf("Sum of %d and %d is %d%n",
9     15, 40, 55);
10
11 /* Character and String */
12 char initial = 'J';
13 String name = "John";
14 System.out.printf("%c. %s%n", initial, name);
```

STRING.FORMAT() USAGE

```
1 /* Basic formatting */
2 int age = 22;
3 String str = String.format("Age: %d", age);
4
5 /* Complex formatting */
6 String details = String.format(
7     "Name: %s%nAge: %d%nHeight: %.2f",
8     "Mike", 25, 1.75
9 );
10
11 /* Using formatted() method (Java 15+) */
12 String result = "Hello %s!".formatted("World");
```

COMMON FORMAT SPECIFIERS

```
1 int number = 42;
2 double price = 19.99;
3 String name = "Alice";
4 char grade = 'A';
5
6 System.out.printf(
7     "Student: %s%nGrade: %c%n" +
8     "Number: %d%nPrice: %.2f%n",
9     name, grade, number, price
10 );
```

WRAPPER CLASSES

PRIMITIVE TYPES AND THEIR WRAPPERS

```
1 /* Primitive type to wrapper mapping */
2 byte primitiveNum = 42;
3 Byte wrapperByte = Byte.valueOf(primitiveNum);
4
5 int count = 100;
6 Integer wrapperInt = Integer.valueOf(count);
7
8 char letter = 'A';
9 Character wrapperChar = Character.valueOf(letter);
10
11 boolean flag = true;
12 Boolean wrapperBool = Boolean.valueOf(flag);
13
14 /* Creating from strings */
15 Integer fromString = Integer.valueOf("123");
16 Double fromDecimal = Double.valueOf("123.45");
```

BOXING AND UNBOXING

```
1 /* Manual boxing and unboxing */
2 int primitive = 100;
3 Integer wrapped = Integer.valueOf(primitive);      /* boxing */
4 int unwrapped = wrapped.intValue();                /* unboxing */
5
6 /* Automatic boxing and unboxing */
7 Double price = 19.99;                            /* autoboxing */
8 double samePrice = price;                        /* auto-unboxing */
9
10 /* Type matching is important */
11 Long validLong = 100L;                          /* OK */
12 /* Long invalidLong = 100; */                     /* Won't compile! */
13
14 /* Careful with null */
15 Integer nullInteger = null;
16 /* int dangerous = nullInteger; */               /* NPE if uncommented */
17 int safe = (nullInteger != null) ? nullInteger : 0; /* Safe approach */
```

WRAPPER COMPARISON AND BEST PRACTICES

```
1 /* Comparing wrapper objects */
2 Integer num1 = Integer.valueOf(100);
3 Integer num2 = Integer.valueOf(100);
4
5 /* Reference comparison */
6 boolean referenceEqual = (num1 == num2);           /* might be false */
7
8 /* Value comparison */
9 boolean valueEqual = num1.equals(num2);             /* true */
10
11 /* Common pitfalls */
12 Long value = null;
13 /* long result = value + 1; */                      /* NPE if uncommented */
14
15 /* Safe arithmetic with null check */
16 Long safe = (value != null) ? value + 1L : 0L;
17
18 /* Using wrapper in collections */
19 import java.util.ArrayList;
20 ArrayList<Integer> numbers = new ArrayList<>();    /* OK */
21 /* ArrayList<int> wrong = new ArrayList<>(); */ /* Won't compile! */
```

ARRAY

Index	0	1	2	3	4
Element	10.8	14.3	13.5	12.1	9.7

DECLARATION, INSTANTIATION, INITIALIZATION

```
1 int[] array; /* declaration form 1 */
2 int array[]; /* declaration form 2: less used in practice */
```

CREATING AN ARRAY WITH SPECIFIED ELEMENTS

```
1 int[] numbers = { 1, 2, 3 }; /* instantiating and initializing an array of 1, 2, 3 */
2 int a = 1, b = 2, c = 3;
3 int[] numbers = { a, b, c }; /* instantiating and initializing an array of 1, 2, 3 */
```

CREATING AN ARRAY USING THE "NEW" KEYWORD

```
1 int[] numbers;          /* declaration */  
2 numbers = new int[n]; /* instantiation and initialization with default values */
```

```
1 int size = 10;  
2 char[] characters = new char[size];  
3  
4 /* It takes an array, start index, end index (exclusive) and the value for filling the array */  
5 Arrays.fill(characters, 0, size / 2, 'A');  
6 Arrays.fill(characters, size / 2, size, 'B');  
7  
8 System.out.println(Arrays.toString(characters)); /* it prints [A, A, A, A, A, B, B, B, B]
```

THE LENGTH OF AN ARRAY

```
1 int[ ] array = { 1, 2, 3, 4 }; /* an array of numbers */
2
3 int length = array.length; /* number of elements of the array */
4
5 System.out.println(length); /* 4 */
```

ACCESSING ELEMENTS

```
1 int[] numbers = new int[3];          /* numbers: [0, 0, 0] */
2 numbers[0] = 1;                     /* numbers: [1, 0, 0] */
3 numbers[1] = 2;                     /* numbers: [1, 2, 0] */
4 numbers[2] = numbers[0] + numbers[1]; /* numbers: [1, 2, 3] */
```

MULTIDIMENSIONAL ARRAY

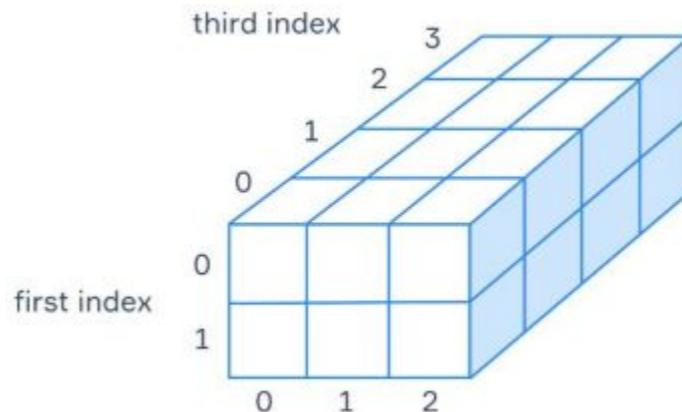
```
1 int[ ][ ] twoDimArray = {  
2     {3, 4, 5}, /* [0] */  
3     {6, 7, 8}, /* [1] */  
4 };
```

twoDimArray[0]	3	4	5
twoDimArray[1]	6	7	8

ITERATING OVER ARRAYS

```
1 /* Iterating over the 2D array */
2 for (int i = 0; i < matrix.length; i++) {
3     for (int j = 0; j < matrix[i].length; j++) {
4         System.out.print(matrix[i][j] + " ");
5     }
6     System.out.println();
7 }
```

MULTIDIMENSIONAL ARRAYS (>2)



```
1 int[][][] threeDimArray = new int[2][3][4];
2
3 int element = 0;
4
5 for (int i = 0; i < threeDimArray.length; i++) {
6     for (int j = 0; j < threeDimArray[i].length; j++) {
7         for (int k = 0; k < threeDimArray[i][j].length; k++) {
8             threeDimArray[i][j][k] = element;
9         }
10        element++;
11    }
12 }
```

ERRORS

EXCEPTION HANDLING

```
1 try {
2     /* code that may throw an exception */
3 } catch (Exception e) {
4     /* code for handling the exception */
5 }
```

GETTING INFO ABOUT AN EXCEPTION

```
1 System.out.println("before the try-catch block"); /* it will be printed */
2
3 try {
4     System.out.println("inside the try block before an exception"); /* it will be printed */
5
6     System.out.println(2 / 0); /* it throws ArithmeticException */
7
8     System.out.println("inside the try block after the exception"); /* it won't be printed */
9 } catch (Exception e) {
10     System.out.println("Division by zero!"); /* it will be printed */
11 }
12
13 System.out.println("after the try-catch block"); /* it will be printed */
```

MULTI-CATCH

```
1 try {
2     /* code that may throw exceptions */
3 } catch (SQLException | IOException e) {
4     /* handling SQLException, IOException and their subclasses */
5     System.out.println(e.getMessage());
6 } catch (Exception e) {
7     /* handling any other exceptions */
8     System.out.println("Something goes wrong");
9 }
```

THE FINALLY BLOCK

```
1 try {
2     /* code that may throw an exception */
3 } catch (Exception e) {
4     /* exception handler */
5 } finally {
6     /* code that will always be executed */
7 }
```

EXAMPLE

```
1 try {
2     System.out.println("inside the try block");
3     Integer.parseInt("101abc"); /* throws a NumberFormatException */
4 } catch (Exception e) {
5     System.out.println("inside the catch block");
6 } finally {
7     System.out.println("inside the finally block");
8 }
9
10 System.out.println("after the try-catch-finally block");
11 // inside the try block
12 // inside the catch block
13 // inside the finally block
14 // after the try-catch-finally block
```

CHECKED EXCEPTIONS

- represented by the Exception class, excluding the RuntimeException subclass

```
1 public static String readLineFromFile() throws FileNotFoundException {
2     Scanner scanner = new Scanner(new File("file.txt")); /* throws java.io.FileNotFoundException */
3     return scanner.nextLine();
4 }
```

UNCHECKED EXCEPTIONS

- represented by the RuntimeException class and all its subclasses

```
1 public static Long convertStringToLong(String str) {  
2     return Long.parseLong(str); /* It may throw a NumberFormatException */  
3 }
```

HIERARCHY OF EXCEPTIONS

THROWING EXCEPTIONS

```
1 public class Main {  
2     public static void main(String args[]) {  
3         RuntimeException exception = new RuntimeException("Something's bad.");  
4         throw exception;  
5     }  
6 }
```

THROWING CHECKED EXCEPTIONS

```
1 public static String readTextFromFile(String path) throws IOException {
2     /* find a file by the specified path */
3
4     if (!found) {
5         throw new IOException("The file " + path + " is not found");
6     }
7
8     /* read and return text from the file */
9 }
```

THROWING UNCHECKED EXCEPTIONS

```
1 class Account {  
2  
3     private long balance = 0;  
4  
5     public void deposit(long amount) {  
6         if (amount <= 0) {  
7             throw new IllegalArgumentException("Incorrect sum " + amount);  
8         }  
9  
10        if (amount >= 100_000_000L) {  
11            throw new IllegalArgumentException("Too large amount");  
12        }  
13  
14        balance += amount;  
15    }  
16  
17    public long getBalance() {  
18        return balance;  
19    }  
20 }
```

CUSTOM EXCEPTIONS

```
1 public class MyAppException extends Exception {  
2  
3     public MyAppException(String msg) {  
4         super(msg);  
5     }  
6  
7     public MyAppException(Exception cause) {  
8         super(cause);  
9     }  
10 }
```

```
1 public static void someMethod() throws MyAppException {  
2     throw new MyAppException("Something bad");  
3 }
```

METHODS

THE SYNTAX OF A METHOD

The diagram illustrates the syntax of a Java method. It shows the following components:

- modifiers**: `public static`
- return type**: `int`
- method name**: `countSeeds`
- list of parameters**: `(int parrotWeight, int parrotAge)`
- body**: The code block enclosed in curly braces, containing:
 - `{`
 - `return parrotWeight / 5 + parrotAge;`
 - `}`

```
public static int countSeeds (int parrotWeight, int parrotAge) {  
    return parrotWeight / 5 + parrotAge;  
}
```

METHOD PARAMETERS

```
1 convertEurosToDollars(double dlrRate, long eur);  
2  
3 countMonthlySpendings(long food, long rent, long fun);  
4  
5 replace(char a, char b);
```

BODY AND RETURN STATEMENT

```
1 public static int countSeeds(int parrotWeight, int parrotAge) {  
2     return parrotWeight / 5 + parrotAge; // it returns an int  
3 }
```

```
1 /* the following method just prints the line, so it returns no value */  
2 public static void printSeedsCount(int seeds) {  
3     System.out.println("Give your parrot " + seeds + "g of seeds per day");  
4 }
```

```
1 public static void isPositive(int num) {  
2     if (num > 0) {  
3         System.out.println("the number is positive");  
4     } else {  
5         return;  
6     }  
7 }
```

WHAT HAPPENS WHEN WE INVOKE A METHOD

```
1 public class Main {  
2     public static int countSeeds(int parrotWeight, int parrotAge) {  
3         return parrotWeight / 5 + parrotAge; /* it returns an int */  
4     }  
5  
6     public static void printSeedsCount(int seeds) {  
7         System.out.println("Give your parrot " + seeds + "g of seeds per day");  
8     }  
9  
10    public static void main(String[] args) {  
11        int myParrotWeight = 100;  
12        int myParrotAge = 3;  
13  
14        /**  
15         * Now myParrotPortion equals 23  
16         * because our method countSeeds, as described above,  
17         * calculates parrotWeight / 5 + parrotAge  
18         */  
19        int myParrotPortion = countSeeds(myParrotWeight, myParrotAge);  
20        printSeedsCount(myParrotPortion);  
21    }  
22 }
```

```
1 Give your parrot 23g of seeds per day
```

METHOD OVERLOADING

```
1 class MathOperation {  
2     /* Overloaded method with two int parameters */  
3     public int add(int a, int b) {  
4         return a + b;  
5     }  
6  
7     /* Overloaded method with three int parameters */  
8     public int add(int a, int b, int c) {  
9         return a + b + c;  
10    }  
11  
12    /* Overloaded method with double parameters */  
13    public double add(double a, double b) {  
14        return a + b;  
15    }  
16}  
17  
18 public class Main {  
19     public static void main(String[] args) {  
20         MathOperation math = new MathOperation();  
21  
22         /* Using the different overloaded methods */  
23         System.out.println(math.add(2, 3));          /* 5 (int, int) */  
24         System.out.println(math.add(2, 3, 4));        /* 9 (int, int, int) */  
25         System.out.println(math.add(2.5, 3.5));       /* 6.0 (double, double) */  
26     }  
27 }
```

OVERLOADING AND CASTING

```
1 public class OverloadingExample {  
2  
3     public static void print(short a) {  
4         System.out.println("short arg: " + a);  
5     }  
6  
7     public static void print(int a) {  
8         System.out.println("int arg: " + a);  
9     }  
10  
11    public static void print(long a) {  
12        System.out.println("long arg: " + a);  
13    }  
14  
15    public static void print(double a) {  
16        System.out.println("double arg: " + a);  
17    }  
18  
19    public static void main(String[ ] args) {  
20        print(100);  
21    }  
22 }
```

CALLING METHODS

BASIC METHOD CALLS

```
1 /* Method declaration */
2 static void calculateVolume(int length, int width, int height) {
3     int volume = length * width * height;
4     System.out.println("Volume: " + volume);
5 }
6
7 /* Different ways to call the method */
8 public class MethodCallExample {
9     public static void main(String[] args) {
10         /* Basic method call */
11         calculateVolume(3, 7, 2);      /* Volume: 42 */
12
13         /* Same method, different parameters */
14         calculateVolume(2, 2, 2);      /* Volume: 8 */
15
16         /* Method with no parameters */
17         printHeader();                /* Just needs empty parentheses */
18     }
19
20     static void printHeader() {
21         System.out.println("== Volume Calculator ==");
22     }
23 }
```

DIFFERENT TYPES OF METHOD CALLS

```
1 /* Built-in static method calls */
2 double number = 79.378;
3 long rounded = Math.round(number);          /* Static method from Math class */
4 boolean isLetter = Character.isLetter('A');  /* Static method from Character */
5
6 /* Instance method calls */
7 String text = "Hello, World!";
8 String lower = text.toLowerCase();           /* Instance method on String object */
9 int length = text.length();                  /* Another instance method */
10
11 /* Multiple parameters with different types */
12 static void printStudent(String name, double score) {
13     System.out.println("Name: " + name);
14     System.out.println("Score: " + score);
15 }
16
17 /* Calling method with multiple parameters */
18 printStudent("John", 3.14);
```

COMMON PATTERNS AND BEST PRACTICES

```
1 /* Method call with stored parameters */
2 int height = 10;
3 int width = 5;
4 int depth = 3;
5 calculateVolume(height, width, depth);
6
7 /* Storing method results */
8 String input = " Hello ";
9 String cleaned = input.trim()          /* Remove whitespace */
10              .toLowerCase()        /* Convert to lowercase */
11              .replace('h', 'H'); /* Replace character */
12
13 /* Invalid calls - won't compile */
14 /* calculateVolume(1, 2); */           /* Wrong number of parameters */
15 /* calculateVolume("3", true, 6); */  /* Wrong parameter types */
16
17 /* Using built-in methods */
18 int max = Math.max(5, 10);           /* Built-in method */
19 double random = Math.random();       /* No parameters needed */
20 char uppercase = Character.toUpperCase('a'); /* Built-in conversion */
```

PASSING ARRAYS TO METHODS

```
1 /* Basic array parameter method */
2 public static void processArray(int[ ] array) {
3     /* do something */
4 }
5
6 /* Method that swaps first and last elements */
7 public static void swapFirstAndLastElements(int[ ] nums) {
8     if (nums.length < 1) {
9         return;
10    }
11
12    int temp = nums[nums.length - 1];
13    nums[nums.length - 1] = nums[0];
14    nums[0] = temp;
15 }
16
17 /* Using the swap method */
18 public static void main(String[ ] args) {
19     int[ ] numbers = { 1, 2, 3, 4, 5 };
20
21     System.out.println(Arrays.toString(numbers)); /* before swapping */
22     swapFirstAndLastElements(numbers);           /* swapping */
23     System.out.println(Arrays.toString(numbers)); /* after swapping */
24 }
```

VARIABLE ARGUMENTS (VARARGS)

```
1 /* Method with varargs parameter */
2 public static void printNumberOfArguments(int... numbers) {
3     System.out.println(numbers.length);
4 }
5
6 /* Different ways to call varargs method */
7 public static void main(String[] args) {
8     printNumberOfArguments(1);                      /* prints 1 */
9     printNumberOfArguments(1, 2);                    /* prints 2 */
10    printNumberOfArguments(1, 2, 3);                /* prints 3 */
11    printNumberOfArguments(new int[] { });           /* prints 0 */
12    printNumberOfArguments(new int[] { 1, 2 });       /* prints 2 */
13 }
```

VARARGS RULES AND RESTRICTIONS

```
1 /* INCORRECT - varargs must be last parameter */
2 public static void method(double... varargs, int a) {
3     /* do something */
4 }
5
6 /* CORRECT - varargs as last parameter */
7 public static void method(int a, double... varargs) {
8     /* do something */
9 }
10
11 /* Example combining regular and varargs parameters */
12 public static void main(String[] args) {
13     method(5, 1.1, 2.2, 3.3);      /* regular parameter then varargs */
14     method(10);                  /* just regular parameter */
15     method(7, 1.1);              /* regular parameter and one vararg */
16 }
```

CLASSES AND MEMBERS

DEFINING CLASSES

```
1  /**
2   * The class is a "blueprint" for patients
3   */
4  class Patient {
5
6      String name;
7      int age;
8      float height;
9 }
```

CREATING OBJECTS

```
1 Patient patient = new Patient();
2 System.out.println(patient.name);          /* it prints null */
3 System.out.println(patient.age);           /* it prints 0 */
```

CREATING MULTIPLE OBJECTS OF THE SAME CLASS

```
1 public class PatientDemo {  
2  
3     public static void main(String[ ] args) {  
4  
5         Patient bob = new Patient();  
6  
7         bob.name = "Bob";  
8         bob.age = 30;  
9         bob.height = 180;  
10  
11        System.out.println(bob.name + " " + bob.age + " " + bob.height);  
12  
13        Patient alice = new Patient();  
14  
15        alice.name = "Alice";  
16        alice.age = 22;  
17        alice.height = 165;  
18  
19        System.out.println(alice.name + " " + alice.age + " " + alice.height);  
20    }  
21 }  
22  
23 class Patient {  
24
```

CONSTRUCTOR

- special methods that initialize a new object of the class
- is invoked when an instance is created using the keyword `new`

USING CONSTRUCTORS

```
1 class Patient {  
2  
3     String name;  
4     int age;  
5     float height;  
6  
7     public Patient(String name, int age, float height) {  
8         this.name = name;  
9         this.age = age;  
10        this.height = height;  
11    }  
12 }
```

```
1 Patient patient1 = new Patient("Slavko", 40, 182.0f);  
2 Patient patient2 = new Patient("Mirko", 33, 171.5f);
```

KEYWORD THIS

```
1 this.name = name;  
2 this.age = age;  
3 this.height = height;
```

DEFAULT AND NO-ARGUMENT CONSTRUCTOR

```
1 class Patient {  
2  
3     String name;  
4     int age;  
5     float height;  
6 }
```

```
1 Patient patient = new Patient();
```

```
1 class Patient {  
2  
3     String name;  
4     int age;  
5     float height;  
6  
7     public Patient() {  
8         this.name = "Unknown";  
9     }  
10 }
```

MULTIPLE CONSTRUCTORS

CONSTRUCTOR OVERLOADING

```
1 public class Robot {  
2     String name;  
3     String model;  
4  
5     public Robot() {  
6         this.name = "Anonymous";  
7         this.model = "Unknown";  
8     }  
9  
10    public Robot(String name, String model) {  
11        this.name = name;  
12        this.model = model;  
13    }  
14 }
```

```
1 Robot anonymous = new Robot(); /* name is "Anonymous", model is "Unknown" */  
2 Robot andrew = new Robot("Andrew", "NDR-114"); /* name is "Andrew", model is "NDR-114" */
```

INVOKING CONSTRUCTORS FROM OTHER CONSTRUCTORS

```
1 this(); /* calls a no-argument constructor */
```

```
1 this("arg1", "arg2"); /* calls a constructor with two string arguments */
```

INVOKING CONSTRUCTORS FROM OTHER CONSTRUCTORS

```
1 public class Robot {  
2     String name;  
3     String model;  
4     int lifetime;  
5  
6     public Robot() {  
7         this.name = "Anonymous";  
8         this.model = "Unknown";  
9     }  
10  
11    public Robot(String name, String model) {  
12        this(name, model, 20);  
13    }  
14  
15    public Robot(String name, String model, int lifetime) {  
16        this.name = name;  
17        this.model = model;  
18        this.lifetime = lifetime;  
19    }  
20 }
```

INVOKING CONSTRUCTORS FROM OTHER CONSTRUCTORS

```
1 public Robot(String name, String model, int lifetime) {  
2     this.name = name;  
3     this.model = model;  
4     this.lifetime = lifetime;  
5     System.out.println("The third constructor is invoked");  
6 }
```

```
1 Robot andrew = new Robot("Andrew", "NDR-114"); /* The third constructor is invoked */
```

INSTANCE METHODS

- do not belong to the class itself, but to objects created from the class.
- can access instance variables (fields) and other instance methods of the class.
- are invoked on an instance (object) of the class, using the object reference.

WHAT'S THE DIFFERENCE?

```
1 class Human {  
2     String name;  
3     int age;  
4  
5     public static void printStatic() {  
6         System.out.println("It's a static method");  
7     }  
8  
9     public void printInstance() {  
10        System.out.println("It's an instance method");  
11    }  
12 }
```

UNDERSTANDING: STATIC AND INSTANCE

```
1 public static void main(String[] args) {  
2  
3     Human.printStatic(); /* will print "It's a static method" */  
4 }
```

```
1 public static void main(String[] args) {  
2  
3     Human peter = new Human();  
4     peter.printInstance(); /* will print "It's an instance method" */  
5  
6     Human alice = new Human();  
7     alice.printInstance(); /* will print "It's an instance method" */  
8 }
```

INSTANCE METHODS AND THIS KEYWORD

```
1 class Human {  
2     String name;  
3     int age;  
4  
5     public static void averageWorking() {  
6         System.out.println("An average human works 40 hours per week.");  
7     }  
8  
9     public void work() {  
10        System.out.println(this.name + " loves working!");  
11    }  
12  
13    public void workTogetherWith(Human other) {  
14        System.out.println(this.name + " loves working with " + other.name + '!');  
15    }  
16 }
```

EXAMPLE

```
1 public static void main(String[ ] args) {
2
3     Human.averageWorking(); /* "An average human works 40 hours per week." */
4
5     Human peter = new Human();
6     peter.name = "Peter";
7     peter.work(); /* "Peter loves working!" */
8
9
10    Human alice = new Human();
11    alice.name = "Alice";
12    alice.work(); /* "Alice loves working!" */
13
14    peter.workTogetherWith(alice); /* "Peter loves working with Alice!" */
15 }
```

STATIC MEMBERS

```
1 class SomeClass {  
2  
3     public static String staticStringField;  
4  
5     public static int staticIntField;  
6 }
```

```
1 SomeClass.staticIntField = 10;  
2 SomeClass.staticStringField = "it's a static member";  
3  
4 System.out.println(SomeClass.staticIntField); /* It prints "10" */  
5 System.out.println(SomeClass.staticStringField); /* It prints "it's a static member" */
```

CLASS CONSTANTS

```
1 class Physics {  
2  
3     /**  
4      * The speed of light in a vacuum (m/s)  
5      */  
6     public static final long SPEED_OF_LIGHT = 299_792_458;  
7  
8     /**  
9      * Electron mass (kg)  
10     */  
11    public static final double ELECTRON_MASS = 9.1093837e-31;  
12 }
```

```
1 System.out.println(Physics.ELECTRON_MASS);      /* 9.1093837E-31 */  
2 System.out.println(Physics.SPEED_OF_LIGHT);       /* 299792458 */
```

OBJECTS AND THEIR PROPERTIES

```
1 class Patient {  
2     String name;  
3     int age;  
4 }
```

```
1 Patient patient = new Patient();
```

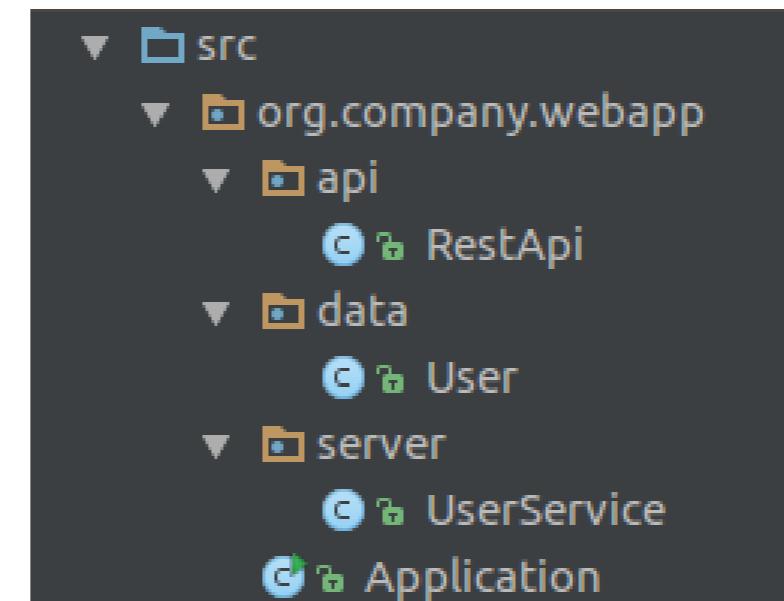
IMMUTABILITY OF OBJECTS

```
1 Patient patient = new Patient();
2
3 patient.name = "Mary";
4 patient.name = "Alice";
```

SHARING REFERENCES

```
1 Patient patient = new Patient();
2
3 patient.name = "Mary";
4 patient.age = 24;
5
6 System.out.println(patient.name + " " + patient.age); /* Mary 24 */
7
8 Patient p = patient;
9
10 System.out.println(p.name + " " + p.age); /* Mary 24 */
```

GROUPING CLASSES WITH PACKAGES



```
1 package org.acme.webapp.data;
2
3 public class User {
4 }
```

IMPORTING CLASSES

```
1 org.acme.java.packages.theory.p1.A  
2 org.acme.java.packages.theory.p2.B
```

```
1 package org.acme.java.packages.theory.p1; /* current package */  
2  
3 import org.acme.java.packages.theory.p2.B; /* it's required to use the import */  
4  
5 public class A {  
6  
7     public static void method() {  
8  
9         B b = new B();  
10    }  
11 }
```

IMPORTING STANDARD CLASSES

- There is no difference between importing standard or custom classes.
- `java.lang` package is always automatically imported.
- This package contains many widely used classes, such as `String`, `System`, `Long`, `Integer`, `NullPointerException` and others.
- the default package:
 - if we do not write a package statement our class goes into default package
 - This package has a big disadvantage, classes located here can't be imported to classes located inside named packages.

ACCESS CONTROL

GETTERS AND SETTERS

- methods used to access and modify the values of private fields (also known as instance variables) in a class
- essential part of encapsulation

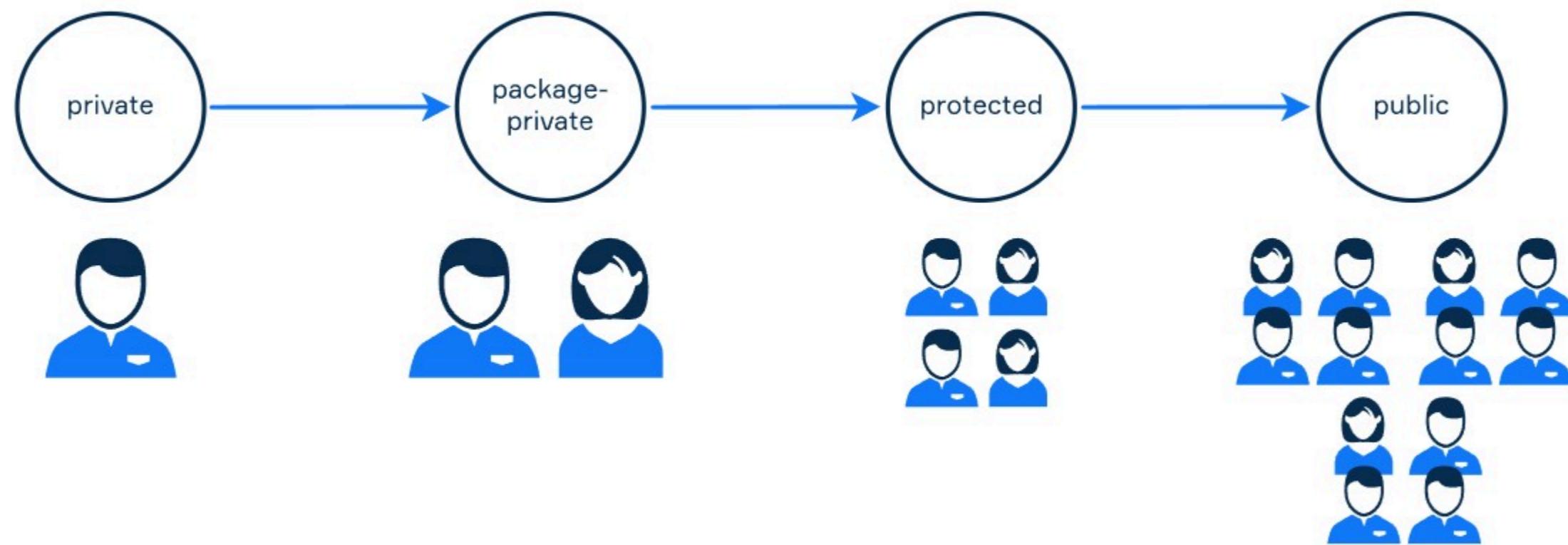
EXAMPLE

```
1 class Account {  
2  
3     private long id;  
4     private String code;  
5     private long balance;  
6     private boolean enabled;  
7  
8     public long getId() {  
9         return id;  
10    }  
11  
12    public void setId(long id) {  
13        this.id = id;  
14    }  
15  
16    public String getCode() {  
17        return code;  
18    }  
19  
20    public void setCode(String code) {  
21        this.code = code;  
22    }  
23  
24    public long getBalance() {  
25        return balance;  
26    }  
27}
```

EXAMPLE

```
1 Account account = new Account();
2
3 account.setId(1000);
4 account.setCode("62968503812");
5 account.setBalance(100_000_000);
6 account.setEnabled(true);
7
8 System.out.println(account.getId());          /* 1000 */
9 System.out.println(account.getCode());         /* 62968503812 */
10 System.out.println(account.getBalance());       /* 100000000 */
11 System.out.println(account.isEnabled());        /* true */
```

ACCESS MODIFIERS



PUBLIC AND PACKAGE-PRIVATE CLASSES

- **package-private** (default, no explicit modifier): visible only for classes from the same package
- **public**: visible to all classes everywhere

```
1 package org.acme.java.packages.theory.p1;  
2  
3 class PackagePrivateClass{  
4  
5 }
```

```
1 package org.acme.java.packages.theory.p2;  
2  
3 public class PublicClass {  
4  
5 }
```

PRIVATE MEMBERS

```
1 public class Counter {  
2     private long current = 0;  
3  
4     public long getCurrent() {  
5         return this.current;  
6     }  
7  
8     public void inc() {  
9         inc(1L);  
10    }  
11  
12  
13     private void inc(long val) {  
14         this.current += val;  
15     }  
16 }  
17 }
```

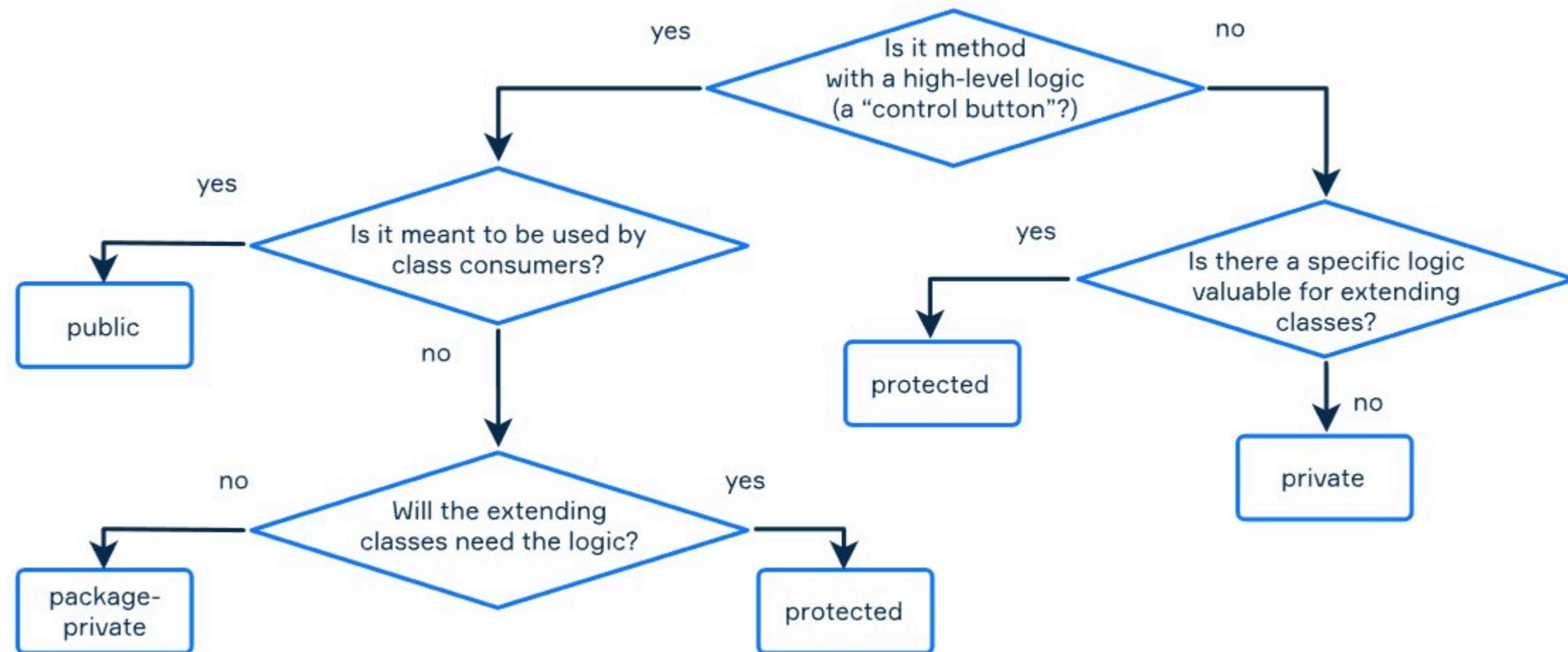
PACKAGE-PRIVATE MEMBERS

```
1 public class Salary {
2     long income;
3
4     Salary(long income) {
5         this.income = income;
6     }
7 }
8
9 public class Promotion {
10    Salary salary;
11
12    Promotion(Salary salary) {
13        this.salary = salary;
14    }
15
16    public void promote() {
17        salary.income += 1500;
18    }
19 }
```

PROTECTED MODIFIER

```
1 package org.acme.bluetooth;
2
3 public class MobileGadget {
4
5     protected void printNotification(String data) {
6         System.out.println(data);
7     }
8 }
```

RECAP



LEVEL OF ACCESS

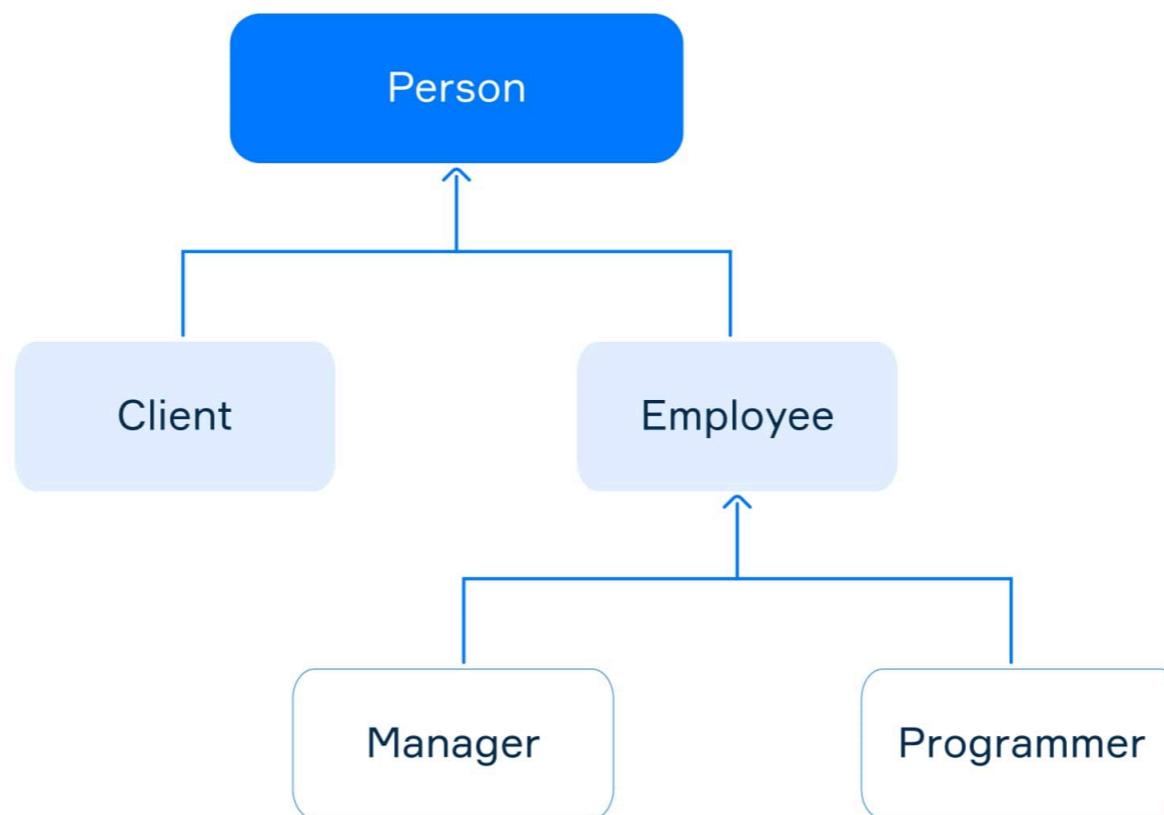
Modifier	Class	Package	Subclass	Global
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

INHERITANCE

BASIC INHERITANCE CONCEPTS

```
1 /* Basic class hierarchy */
2 class SuperClass { }
3
4 class SubClassA extends SuperClass { }
5
6 class SubClassB extends SuperClass { }
7
8 class SubClassC extends SubClassA { }
9
10 /* Final class cannot be extended */
11 final class FinalClass { }
12
13 /* This would cause compile error */
14 /* class SubClass extends FinalClass { } */
```

INHERITANCE EXAMPLE: COMPANY HIERARCHY



```
1 class Person {  
2     protected String name;  
3     protected int yearOfBirth;  
4     protected String address;  
5  
6     /* public getters and setters for all fields here */  
7 }  
8  
9 class Employee extends Person {  
10    protected Date startDate;  
11    protected Long salary;  
12  
13    /* public getters and setters for all fields here */  
14 }  
15  
16 class Programmer extends Employee {  
17     protected String[] programmingLanguages;  
18  
19     public String[] getProgrammingLanguages() {  
20         return programmingLanguages;  
21     }  
22  
23     public void setProgrammingLanguages(String[] programmingLanguages) {  
24         this.programmingLanguages = programmingLanguages;  
25     }  
26 }
```

USING INHERITED MEMBERS

```
1 /* Creating and using a Programmer object */
2 Programmer p = new Programmer();
3
4 p.setName("John Elephant");
5 p.setYearOfBirth(1985);
6 p.setAddress("Some street, 15");
7 p.setStartDate(new Date());
8 p.setSalary(500_000L);
9 p.setProgrammingLanguages(new String[] { "Java", "Scala", "Kotlin" });
10
11 /* Using inherited and class-specific methods */
12 System.out.println(p.getName());      /* inherited from Person */
13 System.out.println(p.getSalary());     /* inherited from Employee */
14 System.out.println(Arrays.toString(p.getProgrammingLanguages()));
15
16 /* Some standard final classes */
17 /* These cannot be extended */
18 final class String { }
19 final class Integer { }
20 final class Math { }
```

THE SUPER KEYWORD

- super keyword accesses parent class members
- Used for fields, methods, and constructors
- Similar to this but refers to parent class
- Essential for inheritance functionality

ACCESSING SUPERCLASS FIELDS AND METHODS

```
1 class SuperClass {
2     protected int field;
3
4     protected int getField() {
5         return field;
6     }
7
8     protected void printBaseValue() {
9         System.out.println(field);
10    }
11 }
12
13 class SubClass extends SuperClass {
14     protected int field; /* Same name as superclass field */
15
16     public SubClass() {
17         this.field = 30; /* Initializes SubClass field */
18         super.field = 20; /* Initializes SuperClass field */
19     }
20
21     public void printSubValue() {
22         super.printBaseValue(); /* Calls superclass method */
23         System.out.println(field);
24     }
25 }
```

INVOKING SUPERCLASS CONSTRUCTOR

```
1 class Person {  
2     protected String name;  
3     protected int yearOfBirth;  
4     protected String address;  
5  
6     public Person(String name, int yearOfBirth, String address) {  
7         this.name = name;  
8         this.yearOfBirth = yearOfBirth;  
9         this.address = address;  
10    }  
11 }  
12  
13 class Employee extends Person {  
14     protected Date startDate;  
15     protected Long salary;  
16  
17     public Employee(String name, int yearOfBirth, String address,  
18                      Date startDate, Long salary) {  
19         super(name, yearOfBirth, address); /* Must be first statement */  
20  
21         this.startDate = startDate;  
22         this.salary = salary;  
23     }  
24 }
```

SUPER KEYWORD RULES

```
1 class Parent {  
2     protected int value = 10;  
3  
4     protected void print() {  
5         System.out.println("Parent: " + value);  
6     }  
7 }  
8  
9 class Child extends Parent {  
10    protected int value = 20; /* Hides parent's value */  
11  
12    public void accessParent() {  
13        System.out.println(super.value); /* Prints 10 */  
14        super.print(); /* Calls parent's print */  
15  
16        /* Must be first in constructor */  
17        /* public Child() {  
18            super(); // OK - first statement  
19            System.out.println("Child");  
20        } */  
21    }  
22 }
```

REFERRING TO SUBCLASS OBJECTS

- Two ways to refer to subclass objects
- Using subclass reference
- Using superclass reference
- Each approach has specific use cases

CLASS HIERARCHY EXAMPLE

```
1 class Person {  
2     protected String name;  
3     protected int yearOfBirth;  
4     protected String address;  
5     /* public getters and setters for all fields */  
6 }  
7  
8 class Client extends Person {  
9     protected String contractNumber;  
10    protected boolean gold;  
11    /* public getters and setters for all fields */  
12 }  
13  
14 class Employee extends Person {  
15     protected Date startDate;  
16     protected Long salary;  
17     /* public getters and setters for all fields */  
18 }  
19  
20 /* Different ways to create objects */  
21 Person person = new Person();      /* reference is Person, object is Person */  
22 Client client = new Client();      /* reference is Client, object is Client */  
23  
24 /* Using superclass reference */  
25 Person employee = new Employee(); /* reference is Person, object is Employee */  
26  
27 /* Invalid references */
```

ACCESSING MEMBERS THROUGH REFERENCES

```
1 /* Using superclass reference */
2 Person employee = new Employee();
3
4 employee.setName("Ginger R. Lee");           /* Ok - Person method */
5 employee.setYearOfBirth(1980);                 /* Ok - Person method */
6 /* employee.setSalary(30000); */                /* Error - Employee method */
7
8 /* Casting between classes */
9 Person person = new Client();
10 Client clientAgain = (Client) person;        /* Ok - object is Client */
11 /* Employee emp = (Employee) person; */        /* ClassCastException */
12
13 /* After casting, can access subclass members */
14 clientAgain.setContractNumber("12345");       /* Now accessible */
```

PRACTICAL USE CASES

```
1 /* Method accepting superclass reference */
2 public static void printNames(Person[ ] persons) {
3     for (Person person : persons) {
4         System.out.println(person.getName());
5     }
6 }
7
8 /* Using the method */
9 Person person = new Employee();
10 person.setName("Ginger R. Lee");
11
12 Client client = new Client();
13 client.setName("Pauline E. Morgan");
14
15 Employee employee = new Employee();
16 employee.setName("Lawrence V. Jones");
17
18 /* Array of different types */
19 Person[ ] persons = {person, client, employee};
20 printNames(persons);
```

RUNTIME TYPE CHECKING

- Determine actual type of object at runtime
- Three main approaches:
 - instanceof operator
 - Pattern matching for instanceof
 - Reflection using getClass()
- Useful before casting objects

THE INSTANCEOF OPERATOR

```
1 class Shape { }
2 class Circle extends Shape { }
3 class Rectangle extends Shape { }
4
5 /* Creating objects with superclass reference */
6 Shape circle = new Circle();
7 Shape rect = new Rectangle();
8
9 /* Basic instanceof checks */
10 boolean circleIsCircle = circle instanceof Circle;      /* true */
11 boolean circleIsRectangle = circle instanceof Rectangle; /* false */
12 boolean circleIsShape = circle instanceof Shape;        /* true */
13
14 /* Testing rectangle object */
15 boolean rectIsRectangle = rect instanceof Rectangle;    /* true */
16 boolean rectIsCircle = rect instanceof Circle;          /* false */
17 boolean rectIsShape = rect instanceof Shape;            /* true */
18
19 /* Won't compile - types not in hierarchy */
20 Circle c = new Circle();
21 /* boolean circleIsRect = c instanceof Rectangle; */ /* Compile error */
```

PATTERN MATCHING WITH INSTANCEOF

```
1 /* Traditional approach */
2 public class PatternMatchingDemo {
3     public static void main(String[ ] args) {
4         Object obj = " ";
5
6         if (obj instanceof String) {
7             String str = (String) obj;
8             if (str.isBlank()) {
9                 System.out.println("Contains only whitespace");
10            }
11        }
12    }
13 }
14
15 /* Using pattern matching (Java 16+) */
16 public class PatternMatchingDemo {
17     public static void main(String[ ] args) {
18         Object obj = " ";
19
20         if (obj instanceof String str && str.length() > 0) {
21             if (str.isBlank()) {
22                 System.out.println("Contains only whitespace");
23             }
24         }
25     }
26 }
```

REFLECTION FOR TYPE CHECKING

```
1 /* Using getClass() method */
2 Shape circle = new Circle();
3
4 boolean equalsCircle = circle.getClass() == Circle.class;      /* true */
5 boolean equalsShape = circle.getClass() == Shape.class;        /* false */
6 boolean rectangle = circle.getClass() == Rectangle.class;     /* false */
7
8 /* Using instanceof() method */
9 boolean instanceofCircle = Circle.class.isInstance(circle);   /* true */
10 boolean instanceofShape = Shape.class.isInstance(circle);     /* true */
11 boolean instanceofRect = Rectangle.class.isInstance(circle);   /* false */
12
13 /* Safe casting example */
14 Shape shape = new Circle();
15 if (shape.getClass() == Circle.class) {
16     Circle circle = (Circle) shape;
17     /* now we can safely use circle methods */
18 }
```

POLYMORPHISM

- polymorphism means that something (an object or another entity) has many forms
- Java provides two types of polymorphism: static (compile-time) and dynamic (run-time) polymorphism

RUNTIME POLYMORPHIC BEHAVIOR

```
1 class MythicalAnimal {  
2  
3     public void hello() {  
4         System.out.println("Hello, I'm an unknown animal");  
5     }  
6 }  
7  
8 class Chimera extends MythicalAnimal {  
9     @Override  
10    public void hello() {  
11        System.out.println("Hello! Hello!");  
12    }  
13 }  
14  
15 class Dragon extends MythicalAnimal {  
16     @Override  
17     public void hello() {  
18         System.out.println("Rrrr...");  
19     }  
20 }
```

```
1 MythicalAnimal chimera = new Chimera();  
2 MythicalAnimal dragon = new Dragon();  
3 MythicalAnimal animal = new MythicalAnimal();  
4  
5 chimera.hello(); /* Hello! Hello! */  
6 dragon.hello(); /* Rrrr... */
```

POLYMORPHISM WITHIN A CLASS HIERARCHY

```
1 class File {  
2  
3     protected String fullName;  
4  
5     /* constructor with a single parameter */  
6     /* getters and setters */  
7  
8     public void printFileInfo() {  
9         String info = this.getFileInfo(); /* here is polymorphic behavior!!! */  
10        System.out.println(info);  
11    }  
12  
13    protected String getFileInfo() {  
14        return "File: " + fullName;  
15    }  
16 }  
17  
18 class ImageFile extends File {  
19  
20     protected int width;  
21     protected int height;  
22     protected byte[] content;  
23  
24     /* constructor  
25        getters and setters */  
26  
27     @Override
```

EXAMPLE

```
1 File img = new ImageFile("/path/to/file/img.png", 480, 640, someBytes); /* assigning an
2
3 img.printFileInfo(); /* It prints "Image: /path/to/file/img.png, width: 480, height: 640
```

METHOD OVERRIDING

OVERRIDING INSTANCE METHODS

```
1 class Mammal {  
2  
3     public String sayHello() {  
4         return "ohllalalalalalaoaoaoa";  
5     }  
6 }  
7  
8 class Cat extends Mammal {  
9  
10    @Override  
11    public String sayHello() {  
12        return "meow";  
13    }  
14 }  
15  
16 class Human extends Mammal {  
17  
18    @Override  
19    public String sayHello() {  
20        return "hello";  
21    }  
22 }
```

EXAMPLE

```
1 Mammal mammal = new Mammal();
2 System.out.println(mammal.sayHello()); /* it prints "ohllalalalalalaaoaoaoa" */
3
4 Cat cat = new Cat();
5 System.out.println(cat.sayHello()); /* it prints "meow" */
6
7 Human human = new Human();
8 System.out.println(human.sayHello()); /* it prints "hello" */
```

RULES FOR OVERRIDING METHODS

- the method must have the same name as in the superclass
- the same arguments as in the superclass method
- the return type should be the same type or a subtype of the return type declared in the method of the superclass
- the access level must be the same or more open
- a private method cannot be overridden because it's not inherited by subclasses
- if the superclass and its subclass are in the same package, then package-private methods can be overridden
- static methods cannot be overridden.

FORBIDDING OVERRIDING

```
1 public final void method() {  
2     /* do something */  
3 }
```

OVERRIDING AND OVERLOADING METHODS TOGETHER

```
1 class SuperClass {  
2  
3     public void invokeInstanceMethod() {  
4         System.out.println("SuperClass: invokeInstanceMethod");  
5     }  
6 }  
7  
8 class SubClass extends SuperClass {  
9  
10    @Override  
11    public void invokeInstanceMethod() {  
12        System.out.println("SubClass: invokeInstanceMethod is overridden");  
13    }  
14  
15    /* @Override -- method doesn't override anything */  
16    public void invokeInstanceMethod(String s) {  
17        System.out.println("SubClass: overloaded invokeInstanceMethod(String)");  
18    }  
19 }
```

```
1 SubClass clazz = new SubClass();  
2  
3 clazz.invokeInstanceMethod();      /* SubClass: invokeInstanceMethod() is overridden */  
4 clazz.invokeInstanceMethod("s"); /* SubClass: overloaded invokeInstanceMethod(String) */
```

INTERFACES AND ABSTRACT CLASSES

INTERFACE

```
1 interface DrawingTool {  
2     void draw(Curve curve);  
3 }
```

```
1 class Pencil implements DrawingTool {  
2     ...  
3     public void draw(Curve curve) {...}  
4 }  
5  
6 class Brush implements DrawingTool {  
7     ...  
8     public void draw(Curve curve) {...}  
9 }
```

DECLARING INTERFACES

```
1 interface Interface {  
2  
3     int INT_CONSTANT = 0; /* it's a constant, the same as public static final int INT_CONSTANT = 0;  
4  
5     void instanceMethod1();  
6  
7     void instanceMethod2();  
8  
9     static void staticMethod() {  
10         System.out.println("Interface: static method");  
11     }  
12  
13    default void defaultMethod() {  
14        System.out.println("Interface: default method");  
15    }  
16  
17    private void privateMethod() {  
18        System.out.println("Interface: private method");  
19    }  
20 }
```

IMPLEMENTING INTERFACES

```
1 class Class implements Interface {  
2  
3     @Override  
4     public void instanceMethod1() {  
5         System.out.println("Class: instance method1");  
6     }  
7  
8     @Override  
9     public void instanceMethod2() {  
10        System.out.println("Class: instance method2");  
11    }  
12 }
```

```
1 Interface instance = new Class();  
2  
3 instance.instanceMethod1(); /* it prints "Class: instance method1" */  
4 instance.instanceMethod2(); /* it prints "Class: instance method2" */  
5 instance.defaultMethod(); /* it prints "Interface: default method. It can be overridden" */
```

DEFAULT METHODS

```
1 interface Feature {  
2     default void action() {  
3         System.out.println("Default action");  
4     }  
5 }
```

```
1 class FeatureImpl implements Feature {  
2     public void action() {  
3         System.out.println("FeatureImpl-specific action");  
4     }  
5 }  
6 ...  
7  
8 Feature feature = new FeatureImpl();  
9 feature.action(); // FeatureImpl-specific action
```

WHY ARE THEY NEEDED

```
1 interface Movable {  
2     void stepAhead();  
3     void turnLeft();  
4     void turnRight();  
5  
6     default void turnAround() {  
7         turnLeft();  
8         turnLeft();  
9     }  
10 }
```

```
1 class Batman implements Movable {  
2     public void stepAhead() {...}  
3     public void turnLeft() {...}  
4     public void turnRight() {...}  
5     public void turnAround() {  
6         turnRight();  
7         turnRight();  
8     }  
9 }
```

THE DIAMOND PROBLEM

- The problem in which a class implements different interfaces that have a default method with the same signature

```
1 interface Jumpable {  
2     void jump();  
3     void turnLeftJump();  
4     void turnRightJump();  
5     default void turnAround() {  
6         turnLeftJump();  
7         turnLeftJump();  
8     }  
9 }
```

```
1 class Spiderman implements Movable, Jumpable {  
2     ...  
3     public void turnAround() {  
4         Movable.super.turnAround();  
5     }  
6 }
```

COMPARABLE

```
1 public class Car implements Comparable<Car> {
2
3     private int number;
4     private String model;
5     private String color;
6     private int weight;
7
8     /* constructor
9      getters, setters */
10
11    @Override
12    public int compareTo(Car otherCar) {
13        return Integer.valueOf(getNumber()).compareTo(otherCar.getNumber());
14    }
15
16 }
```

IMPLEMENTING THE COMPARETO METHOD

To implement it correctly we need to make sure that the method returns:

- A positive integer (for example, 1), if the current object is greater
- A negative integer (for example, -1), if the current object is less
- Zero, if they are equal

```
1 @Override
2 public int compareTo(Integer anotherInteger) {
3     return compare(this.value, anotherInteger.value);
4 }
5
6 public static int compare (int x, int y) {
7     return (x < y) ? -1 : ((x == y) ? 0 : 1);
8 }
```

EXAMPLE

```
1 class Coin implements Comparable<Coin> {
2     private final int nominalValue;      /* nominal value */
3     private final int mintYear;          /* the year the coin was minted */
4
5     Coin(int nominalValue, int mintYear) {
6         this.nominalValue = nominalValue;
7         this.mintYear = mintYear;
8     }
9     @Override
10    public int compareTo(Coin other) {
11        if (nominalValue == other.nominalValue) {
12            return 0;
13        } else if (nominalValue < other.nominalValue) {
14            return -1;
15        } else {
16            return 1;
17        }
18    }
19    @Override
20    public String toString() {
21        return "Coin{nominal=" + nominalValue + ", year=" + mintYear + "}";
22    }
23    /* getters, setters, hashCode and toString */
24 }
```

EXAMPLE

```
1 List<Coin> coins = new ArrayList<>();  
2  
3 coins.add(big);  
4 coins.add(medium1);  
5 coins.add(medium2);  
6 coins.add(small);  
7  
8 Collections.sort(coins);  
9 coins.forEach(System.out::println);
```

```
1 Coin{nominal=2, year=2000}  
2 Coin{nominal=10, year=2016}  
3 Coin{nominal=10, year=2001}  
4 Coin{nominal=25, year=2006}
```

ABSTRACT CLASS

- declared with the keyword `abstract`
- represents an abstract concept that is used as a base class for subclasses

EXAMPLE

```
1 public abstract class Pet {  
2  
3     protected String name;  
4     protected int age;  
5  
6     protected Pet(String name, int age) {  
7         this.name = name;  
8         this.age = age;  
9     }  
10  
11    public abstract void say(); /* an abstract method */  
12 }
```

```
1 Pet pet = new Pet("Unnamed", 5); /* this throws a compile time error */
```

SUBCLASSES

```
1 class Cat extends Pet {  
2  
3     /* It can have additional fields as well */  
4  
5     public Cat(String name, int age) {  
6         super(name, age);  
7     }  
8  
9     @Override  
10    public void say() {  
11        System.out.println("Meow!");  
12    }  
13}  
14  
15 class Dog extends Pet {  
16  
17     /* It can have additional fields as well */  
18  
19     public Dog(String name, int age) {  
20         super(name, age);  
21     }  
22  
23     @Override  
24     public void say() {  
25         System.out.println("Woof!");  
26     }  
27}
```

ABSTRACT CLASS VS INTERFACE

The Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance .	Interface supports multiple inheritance .
3) Abstract class can have final, non-final, static and non-static variables .	Interface has only static and final variables .
4) Abstract class can provide the implementation of interface .	Interface can't provide the implementation of abstract class .
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword ? extends?.	An interface class can be implemented using keyword ? implements?.

USING ABSTRACT CLASSES AND INTERFACES TOGETHER

```
1 interface ManagedDevice {  
2  
3     void on();  
4     void off();  
5 }  
6 abstract class AbstractDevice implements ManagedDevice {  
7     protected String serialNumber;  
8     protected boolean on;  
9  
10    public AbstractDevice(String serialNumber) {  
11        this.serialNumber = serialNumber;  
12    }  
13    protected void setOn(boolean on) {  
14        this.on = on;  
15    }  
16 }
```

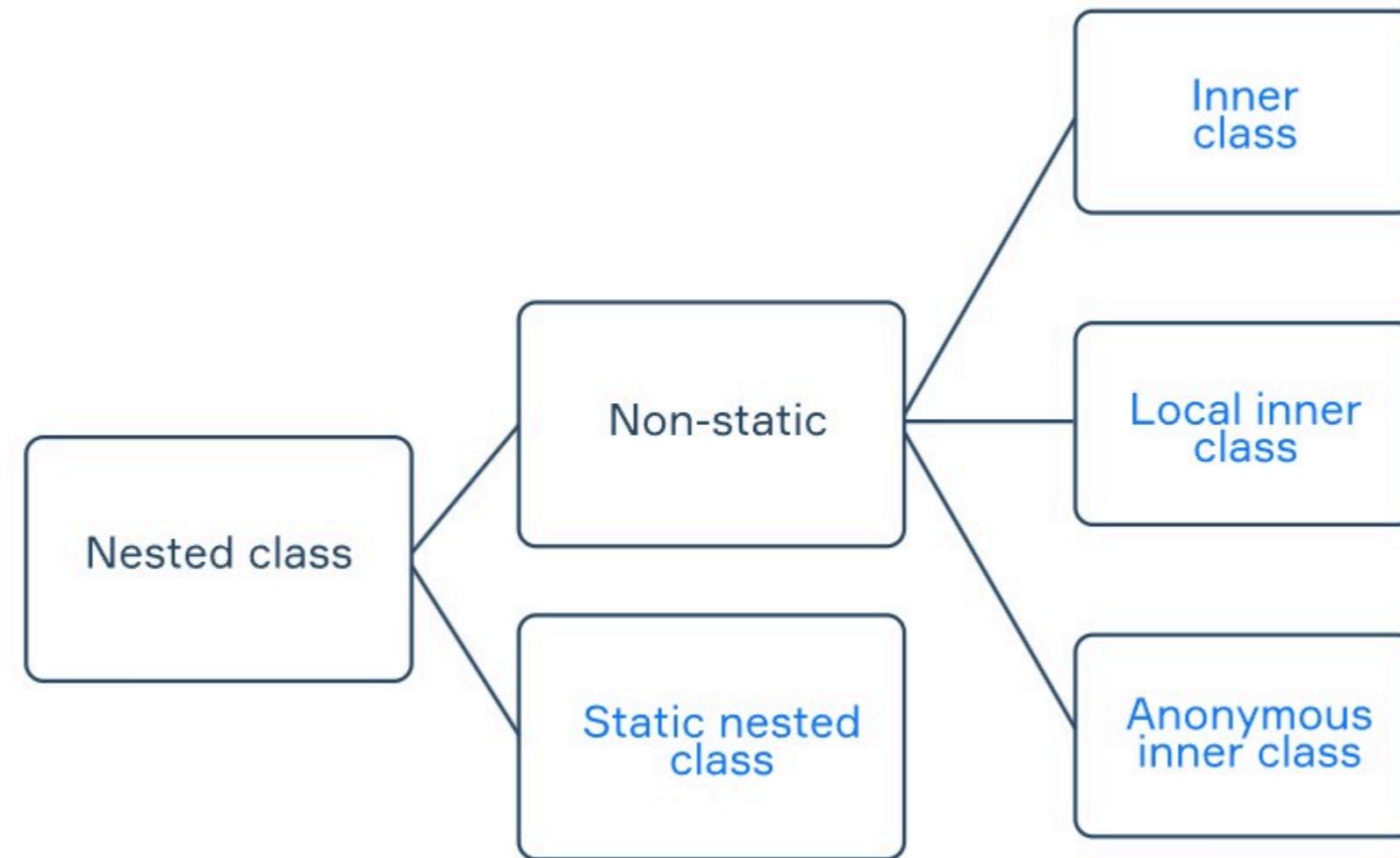
```
1 class Kettle extends AbstractDevice {
2     protected double volume;
3
4     public Kettle(String serialNumber, double volume) {
5         super(serialNumber);
6         this.volume = volume;
7     }
8     @Override
9     public void on() {
10        /* do complex logic to activate all electronic components */
11        setOn(true);
12    }
13    @Override
14    public void off() {
15        /* do complex logic to stop all electronic components */
16        setOn(false);
17    }
18 }
```

NESTED CLASSES

WHAT IS A NESTED CLASS?

```
1 class Superhero {  
2  
3     class MagicCloak {  
4  
5         }  
6  
7     class Hammer {  
8  
9         }  
10 }
```

TYPES OF NESTED CLASSES



INNER CLASS

```
1 public class Cat {  
2     private String name;  
3     public Cat(String name) {  
4         this.name = name;  
5     }  
6     public class Bow {  
7         private String color;  
8         public Bow(String color) {  
9             this.color = color;  
10        }  
11        public void printColor() {  
12            System.out.println("Cat " + Cat.this.name + " has a " + this.color + " bow."  
13        }  
14    }  
15 }
```

```
1 public class Main {  
2     public static void main(String[] args) {  
3         Cat cat = new Cat("Bob");  
4         Cat.Bow bow = cat.new Bow("red");  
5         bow.printColor(); /* Cat Bob has a red bow. */  
6     }  
7 }
```

SCOPE OF THE INNER CLASS

```
1 public class Cat {  
2     private String name;  
3  
4     public Cat(String name) {  
5         this.name = name;  
6     }  
7     private void sayMeow() {  
8         System.out.println(this.name + " says: \"Meow\".");  
9     }  
10    public class Bow {  
11        String color;  
12  
13        public Bow(String color) {  
14            this.color = color;  
15        }  
16        public void putOnABow() {  
17            Cat.this.sayMeow();  
18            System.out.println("Bow is on!");  
19        }  
20        public void printColor() {  
21            System.out.println("Cat " + Cat.this.name + " has a " + this.color + " bow.\\".  
22        }  
23    }  
24 }
```

EXAMPLE

```
1 Cat cat = new Cat("Princess");
2 Cat.Bow bow = cat.new Bow("golden");
3
4 bow.printColor();
5 bow.putOnABow();
```

```
1 Cat Princess has a golden bow.
2
3 Princess says: "Meow".
4 Bow is on!
```

RULES FOR INNER CLASSES

```
1 Outer outer = new Outer();
2 Outer.InnerClass inner = outer.new InnerClass();
```

LOCAL INNER CLASS

```
1 public class Outer {  
2  
3     private int number = 10;  
4  
5     void someMethod() {  
6  
7         class LocalInner {  
8  
9             private void print() {  
10                 System.out.println("number = " + Outer.this.number);  
11             }  
12         }  
13  
14         LocalInner inner = new LocalInner();  
15         inner.print();  
16     }  
17  
18     public static void main(String[ ] args) {  
19         Outer outer = new Outer();  
20         outer.someMethod();  
21     }  
22 }
```

SCOPE OF A LOCAL INNER CLASS

```
1 public class Outer {  
2  
3     private int number = 10;  
4  
5     void someMethod() {  
6         final int x = 5;  
7  
8         class Inner {  
9             private void print() {  
10                 System.out.println("x = " + x);  
11                 System.out.println("number = " + Outer.this.number);  
12             }  
13         }  
14  
15         Inner inner = new Inner();  
16         inner.print();  
17     }  
18  
19     public static void main(String[] args) {  
20         Outer outer = new Outer();  
21         outer.someMethod();  
22     }  
23 }
```

ANONYMOUS CLASS

- enable you to declare and instantiate a class at the same time

```
1 new SuperClassOrInterfaceName() {  
2  
3     // fields  
4  
5     // overridden methods  
6 };
```

WRITING ANONYMOUS CLASSES

```
1 interface SpeakingEntity {  
2  
3     void sayHello();  
4  
5     void sayBye();  
6 }
```

```
1 SpeakingEntity englishSpeakingPerson = new SpeakingEntity() {  
2  
3     @Override  
4     public void sayHello() {  
5         System.out.println("Hello!");  
6     }  
7  
8     @Override  
9     public void sayBye() {  
10        System.out.println("Bye!");  
11    }  
12};
```

STATIC NESTED CLASS

```
1 public class Painting {  
2     private String name;  
3     private static double length;  
4     private static double width;  
5  
6     public static void setLength(double length) {  
7         Painting.length = length;  
8     }  
9     public static void setWidth(double width) {  
10        Painting.width = width;  
11    }  
12    public static class Sketch {  
13        private int id;  
14  
15        public Sketch(int id) {  
16            this.id = id;  
17        }  
18        public void drawSketch() {  
19            drawForest();  
20            drawBear();  
21        }  
22        private void drawForest() {  
23            if (Painting.length > 5 && Painting.width > 3) {  
24                System.out.println("Big forest was drawn in a sketch!");  
25            } else {  
26                System.out.println("Small forest was drawn in a sketch!");  
27            }  
28        }  
29    }  
30}
```

SCOPE OF A STATIC NESTED CLASS

- access to private static fields from a static nested class
- can't see instance variables and methods of an outer class, including the field name in our example
- if you make a static nested class private, then it can only be accessed inside the outer class

DESIGN PATTERNS

- Repeatable solutions to common development problems
- Make code more elegant, flexible, and reusable
- Language-independent concepts
- Essential for well-structured object-oriented architecture

WHAT ARE DESIGN PATTERNS

```
1 /* Example of Singleton pattern */
2 public class Singleton {
3     private static Singleton instance;
4
5     private Singleton() { }
6
7     public static Singleton getInstance() {
8         if (instance == null) {
9             instance = new Singleton();
10        }
11        return instance;
12    }
13 }
14
15 /* Example of pattern structure */
16 class PatternExample {
17     /* Problem: Need single instance */
18     /* Solution: Private constructor + static method */
19     /* When to use: Single shared resource */
20     /* Consequences: Global state, controlled access */
21 }
```

BENEFITS OF DESIGN PATTERNS

```
1 /* Without pattern - complex creation logic */
2 class ComplexObject {
3     private Type1 field1;
4     private Type2 field2;
5
6     public ComplexObject(Parameter1 p1, Parameter2 p2) {
7         /* Complex initialization code */
8     }
9 }
10
11 /* With Builder pattern - cleaner creation */
12 class ComplexObject {
13     public static class Builder {
14         private Type1 field1;
15         private Type2 field2;
16
17         public Builder withField1(Type1 value) {
18             this.field1 = value;
19             return this;
20         }
21
22         public ComplexObject build() {
23             return new ComplexObject(this);
24         }
25     }
26 }
```

IMPORTANT CONSIDERATIONS

```
1  /* Over-complicated example */
2  public class HelloWorld {
3      private final MessageFactory messageFactory;
4      private final MessageDecorator decorator;
5
6      public HelloWorld(MessageFactory factory,
7                          MessageDecorator decorator) {
8          this.messageFactory = factory;
9          this.decorator = decorator;
10     }
11
12     public void greet() {
13         String message = messageFactory.createMessage();
14         System.out.println(decorator.decorate(message));
15     }
16 }
17
18 /* Simple solution is sometimes better */
19 public class HelloWorld {
20     public void greet() {
21         System.out.println("Hello, World!");
22     }
23 }
```

CREATIONAL DESIGN PATTERNS

- Focus on object creation mechanisms
- Separate system functions from object creation
- Make systems more flexible
- Provide alternatives to hard coding

WHEN TO USE CREATIONAL PATTERNS

```
1 /* Example scenario: Multiple related objects */
2 interface Animal { }
3 class Dog implements Animal { }
4 class Cat implements Animal { }
5
6 /* Factory Method Pattern */
7 abstract class AnimalFactory {
8     abstract Animal createAnimal();
9 }
10
11 class DogFactory extends AnimalFactory {
12     Animal createAnimal() {
13         return new Dog();
14     }
15 }
16
17 /* Singleton Pattern */
18 class ConfigurationManager {
19     private static ConfigurationManager instance;
20
21     private ConfigurationManager() { }
22
23     public static ConfigurationManager getInstance() {
24         if (instance == null) {
25             instance = new ConfigurationManager();
26         }
27         return instance;
28     }
29 }
```

TYPES OF CREATIONAL PATTERNS

```
1 /* Factory Method Pattern */
2 interface Product { }
3 class ConcreteProduct implements Product { }
4
5 abstract class Creator {
6     abstract Product factoryMethod();
7 }
8
9 /* Builder Pattern */
10 class Computer {
11     private String cpu;
12     private String memory;
13
14     public static class Builder {
15         private Computer computer = new Computer();
16
17         public Builder setCPU(String cpu) {
18             computer.cpu = cpu;
19             return this;
20         }
21
22         public Computer build() {
23             return computer;
24         }
25     }
26 }
27 }
```

CONSIDERATIONS AND LIMITATIONS

```
1 /* Potential overuse example */
2 class SimpleMessage {
3     /* Without pattern - simple direct creation */
4     String getMessage() {
5         return "Hello";
6     }
7 }
8
9 /* Over-engineered with patterns */
10 interface MessagePrototype { }
11 abstract class MessageFactory { }
12 class MessageBuilder { }
13 class MessageSingleton { }
14
15 class ComplexMessage {
16     private final MessageFactory factory;
17     private final MessageBuilder builder;
18
19     /* Unnecessary complexity for simple task */
20     String getMessage() {
21         return builder.withFactory(factory)
22             .withPrototype(new MessagePrototype())
23             .build()
24             .toString();
25     }
26 }
```

EQUALS AND HASHCODE

- Methods inherited from Object class
- Default implementations insufficient for custom objects
- equals() checks reference equality
- hashCode() provides object identity

EXAMPLE CLASS

```
1 class Person {  
2     private String firstName;  
3     private String lastName;  
4     private int age;  
5  
6     public Person(String firstName, String lastName, int age) {  
7         this.firstName = firstName;  
8         this.lastName = lastName;  
9         this.age = age;  
10    }  
11 }
```

IMPLEMENTATION EXAMPLE

```
1 /* Using default implementations */
2 Person p1 = new Person("John", "Smith", 31);
3 Person p2 = new Person("John", "Smith", 31);
4
5 System.out.println(p1.equals(p2));          /* false - compares references */
6 System.out.println(p1.hashCode());          /* 242131142 */
7 System.out.println(p2.hashCode());          /* 1782113663 */
8
9 /* Compare with String class behavior */
10 String s1 = new String("John Smith");
11 String s2 = new String("John Smith");
12 System.out.println(s1.equals(s2));          /* true */
13 System.out.println(s1.hashCode());          /* same value */
14 System.out.println(s2.hashCode());          /* same value */
```

IMPLEMENTING EQUALS()

```
1 class Person {  
2     private String firstName;  
3     private String lastName;  
4     private int age;  
5  
6     @Override  
7     public boolean equals(Object other) {  
8         /* Check this and other refer to same object */  
9         if (this == other) {  
10             return true;  
11         }  
12  
13         /* Check other is Person and not null */  
14         if (!(other instanceof Person)) {  
15             return false;  
16         }  
17  
18         Person person = (Person) other;  
19  
20         /* Compare all required fields */  
21         return age == person.age &&  
22                 Objects.equals(firstName, person.firstName) &&  
23                 Objects.equals(lastName, person.lastName);  
24     }  
25 }  
26  
27 /* Testing equals implementation */
```

IMPLEMENTING HASHCODE()

```
1 class Person {  
2     private String firstName;  
3     private String lastName;  
4     private int age;  
5  
6     @Override  
7     public int hashCode() {  
8         int result = 17;  
9         result = 31 * result + (firstName == null ? 0 : firstName.hashCode());  
10        result = 31 * result + (lastName == null ? 0 : lastName.hashCode());  
11        result = 31 * result + age;  
12        return result;  
13    }  
14  
15    /* Alternative using Java 7+ utility */  
16    @Override  
17    public int hashCode() {  
18        return Objects.hash(firstName, lastName, age);  
19    }  
20 }
```

TESTING HASHCODE IMPLEMENTATION

```
1 Person p1 = new Person("John", "Smith", 31);
2 Person p2 = new Person("John", "Smith", 31);
3 Person p3 = new Person("Marry", "Smith", 30);
4
5 System.out.println(p1.hashCode()); /* same for p1 and p2 */
6 System.out.println(p2.hashCode()); /* same for p1 and p2 */
7 System.out.println(p3.hashCode()); /* different */
```

JAVA COLLECTIONS

- Collections are containers for grouping elements
- More flexible than arrays
- Automatically handle resizing
- Provide rich set of methods
- Represent different data structures

ARRAYS VS COLLECTIONS

```
1 /* Array limitations */
2 String[ ] array = new String[3]; /* Fixed size */
3 array[0] = "first";
4 /* Need manual resize to add more elements */
```

ARRAYLIST FLEXIBILITY

```
1 ArrayList<String> list = new ArrayList<>();
2
3 list.add("first");          /* Automatic resizing */
4 list.add("second");
5 list.add("third");
6
7 System.out.println(list); /* [first, second, third] */
8
9 list.remove("first");      /* Easy element removal */
10 System.out.println(list); /* [second, third] */
11
12 System.out.println(list.size()); /* 2 */
```

COLLECTION FEATURES

```
1 /* Generic type collection */
2 ArrayList<String> strings = new ArrayList<>();
3 ArrayList<Integer> numbers = new ArrayList<>(); /* wrapper type */
4
5 /* Cannot use primitives directly */
6 /* ArrayList<int> wrong; */ /* Won't compile */
```

CUSTOM CLASS IN COLLECTION

```
1 class Person {  
2     private String name;  
3     ...  
4 }  
5  
6 ArrayList<Person> people = new ArrayList<>();  
7 people.add(new Person("John"));  
8  
9 /* Collection methods */  
10 list.get(0);           /* Access element */  
11 list.add("element"); /* Add element */  
12 list.remove(0);       /* Remove element */  
13 list.size();          /* Get size */
```

GETTING STARTED WITH COLLECTIONS 1

```
1 /* Import collection class */
2 import java.util.ArrayList;
3
4 /* Creating and using collection */
5 ArrayList<String> tasks = new ArrayList<>();
6
7 /* Adding elements */
8 tasks.add("Learn collections");
9 tasks.add("Practice coding");
10 tasks.add("Write tests");
```

GETTING STARTED WITH COLLECTIONS 2

```
1 /* Accessing elements */
2 String firstTask = tasks.get(0);
3 System.out.println(firstTask); /* Learn collections */
4
5 /* Removing elements */
6 tasks.remove("Write tests");
7
8 /* Check size */
9 int count = tasks.size();
10
11 /* Print entire collection */
12 System.out.println(tasks); /* [Learn collections, Practice coding] */
```

CREATING ARRAYLIST

```
1 /* Import the class */
2 import java.util.ArrayList;
3
4 /* Different ways to create ArrayList */
5 ArrayList<String> list1 = new ArrayList<>();      /* Empty, capacity 10 */
6 ArrayList<String> list2 = new ArrayList<>(50);    /* Empty, capacity 50 */
7 ArrayList<String> list3 = new ArrayList<>(anotherList); /* Copy elements */
```

ADDING ELEMENTS TO ARRAYLIST

```
1 ArrayList<String> names = new ArrayList<>();
2 names.add("Justin");                      /* [Justin] */
3 names.add("Helen");                       /* [Justin, Helen] */
4 names.add(1, "Joshua");                   /* [Justin, Joshua, Helen] */
5 names.add(0, "Laura");                    /* [Laura, Justin, Joshua, Helen] */
6
7 /* Replacing elements */
8 names.set(3, "Marie");                  /* [Laura, Justin, Joshua, Marie] */
```

BASIC ARRAYLIST OPERATIONS 1

```
1 /* Creating and adding elements */
2 ArrayList<String> names = new ArrayList<>();
3 names.add("Justin");           /* Add to end */
4 names.add(0, "Laura");        /* Add at index */
5
6 /* Accessing elements */
7 System.out.println(names.size());      /* Get size */
8 System.out.println(names.get(0));       /* Get by index */
9
10 /* Removing elements */
11 names.remove("Justin");            /* Remove by value */
12 names.remove(0);                  /* Remove by index */
13 names.clear();                   /* Remove all */
```

BASIC ARRAYLIST OPERATIONS 2

```
1 /* Additional operations */
2 ArrayList<Integer> numbers = new ArrayList<>();
3 numbers.add(1);
4 numbers.add(2);
5 numbers.add(1);
6
7 System.out.println(numbers.contains(2));      /* true */
8 System.out.println(numbers.indexOf(1));        /* 0 */
9 System.out.println(numbers.lastIndexOf(1));    /* 2 */
```

ITERATING OVER ARRAYLIST

```
1 /* Creating and populating ArrayList */
2 ArrayList<Long> powersOfTen = new ArrayList<>();
3
4 /* Using for loop */
5 for (int i = 0; i < 5; i++) {
6     long power = (long) Math.pow(10, i);
7     powersOfTen.add(power);
8 }
9
10 /* Using for-each loop */
11 for (Long value : powersOfTen) {
12     System.out.print(value + " ");
13 }
14 /* Output: 1 10 100 1000 10000 */
```

ADDING COLLECTIONS

```
1 ArrayList<Integer> numbers1 = new ArrayList<>();
2 numbers1.add(1);
3 numbers1.add(2);
4
5 ArrayList<Integer> numbers2 = new ArrayList<>();
6 numbers2.add(100);
7 numbers2.addAll(numbers1);      /* [100, 1, 2] */
```

COLLECTIONS FRAMEWORK

- Unified architecture for collections
- Part of `java.util` package
- Common data structures and interfaces
- Enables implementation-independent usage
- Includes interfaces, implementations, algorithms

COLLECTION HIERARCHY

COMMON COLLECTION METHODS

```
1 /* Creating collection */
2 Collection<String> collection = new ArrayList<>();
3
4 /* Basic operations */
5 collection.add("Element");           /* Add element */
6 collection.remove("Element");        /* Remove element */
7 collection.size();                  /* Get size */
8 collection.isEmpty();               /* Check if empty */
9 collection.contains("Element");     /* Check containment */
10 collection.clear();                /* Remove all */
```

```
1 /* Working with other collections */
2 Collection<String> other = new ArrayList<>();
3 collection.removeAll(other);           /* Remove common elements */
4
5 /* Iteration methods */
6 for (String element : collection) {    /* for-each loop */
7     System.out.println(element);
8 }
9
10 /* Using forEach (Java 8+) */
11 collection.forEach(System.out::println);
12 collection.forEach(elem -> System.out.println(elem));
```

COLLECTION FEATURES 1

```
1 /* Mutable collection example */
2 Collection<String> mutable = new ArrayList<>();
3 mutable.add("Can add");
4 mutable.remove("Can remove");
5
6 /* Immutable collection (will throw exception) */
7 /* collection.add("Will throw exception"); */
8 /* collection.remove("Will throw exception"); */
```

COLLECTION FEATURES 2

```
1 /* Advanced removal with predicate */
2 Collection<String> words = new ArrayList<>();
3 words.add("English");
4 words.add("Español");
5 words.add("Deutsch");
6
7 /* Remove words starting with 'E' */
8 words.removeIf(word -> word.startsWith("E"));
```

COLLECTIONS

THE LIST INTERFACE

- `E set(int index, E element)` replaces the element at the specified position in this list with the specified element and returns the element that was replaced;
- `E get(int index)` returns the element at the specified position in the list;
- `int indexOf(Object obj)` returns the index of the first occurrence of the element in the list or `-1` if there is no such element;
- `int lastIndexOf(Object obj)` returns the index of the last occurrence of the element in the list or `-1` if there is no such element;
- `List<E> subList(int fromIndex, int toIndex)` returns a sublist of this list from `fromIndex` included to `toIndex` excluded.

IMMUTABLE LISTS

```
1 List<String> emptyList = List.of(); /* 0 elements */
2 List<String> names = List.of("Larry", "Kenny", "Sabrina"); /* 3 elements */
3 List<Integer> numbers = List.of(0, 1, 1, 2, 3, 5, 8, 13); /* 8 elements */
```

```
1 List<String> daysOfWeek = List.of(
2     "Monday",
3     "Tuesday",
4     "Wednesday",
5     "Thursday",
6     "Friday",
7     "Saturday",
8     "Sunday"
9 );
10
11 System.out.println(daysOfWeek.size()); /* 7 */
12 System.out.println(daysOfWeek.get(1)); /* Tuesday */
13 System.out.println(daysOfWeek.indexOf("Sunday")); /* 6 */
14
15 List<String> weekDays = daysOfWeek.subList(0, 5);
16 System.out.println(weekDays); // [Monday, Tuesday, Wednesday, Thursday, Friday]
```

IMMUTABLE LISTS

- Only methods that do not change the elements in the list will work and others will throw an exception:

```
1 daysOfWeek.set(0, "Funday"); /* throws UnsupportedOperationException */
2 daysOfWeek.add("Holiday");   /* throws UnsupportedOperationException */
```

MUTABLE LISTS

```
1 List<Integer> numbers = new ArrayList<>();
2
3 numbers.add(15);
4 numbers.add(10);
5 numbers.add(20);
6
7 System.out.println(numbers); /* [15, 10, 20] */
8
9 numbers.set(0, 30); /* no exceptions here */
10
11 System.out.println(numbers); /* [30, 10, 20] */
```

MUTABLE LISTS

```
1 List<String> immutableList = List.of("one", "two", "three");  
2 List<String> mutableList = new ArrayList<>(immutableList);
```

```
1 List<Integer> numbers = new LinkedList<>();  
2  
3 numbers.add(10);  
4 numbers.add(20);  
5 numbers.add(30);  
6  
7 System.out.println(numbers); // [10, 20, 30]
```

ITERATING OVER A LIST

```
1 List<String> names = List.of("Larry", "Kenny", "Sabrina");
```

1.Using the "for-each" loop:

```
1 /* print every name */
2 for (String name : names) {
3     System.out.println(name);
4 }
```

2.Using indexes and the `size()` method:

```
1 /* print every second name */
2 for (int i = 0; i < names.size(); i += 2) {
3     System.out.println(names.get(i));
4 }
```

LIST EQUALITY

```
1 Objects.equals(List.of(1, 2, 3), List.of(1, 2, 3));      /* true */
2 Objects.equals(List.of(1, 2, 3), List.of(1, 3, 2));      /* false */
3 Objects.equals(List.of(1, 2, 3), List.of(1, 2, 3, 1)); /* false */
4
5 List<Integer> numbers = new ArrayList<>();
6
7 numbers.add(1);
8 numbers.add(2);
9 numbers.add(3);
10
11 Objects.equals(numbers, List.of(1, 2, 3)); /* true */
```

THE SET INTERFACE

- a collection that does not allow duplicate elements
- a subtype of the `Collection` interface
- can be implemented by different classes like `HashSet` , `LinkedHashSet` , and `TreeSet`

IMMUTABLE SETS

```
1 Set<String> emptySet = Set.of();  
2 Set<String> people = Set.of("Larry", "Kenny", "Sabrina");  
3 Set<Integer> numbers = Set.of(100, 200, 300, 400);
```

```
1 System.out.println(emptySet); /* [ ] */  
2 System.out.println(people); /* [Kenny, Larry, Sabrina] or another order */  
3 System.out.println(numbers); /* [400, 200, 300, 100] or another order */
```

```
1 System.out.println(emptySet.contains("hello")); /* false */  
2 System.out.println(people.contains("Sabrina")); /* true */  
3 System.out.println(people.contains("John")); /* false */  
4 System.out.println(numbers.contains(300)); /* true */
```

HASHSET

- represents a set backed by a hash table

```
1 Set<String> countries = new HashSet<>();  
2  
3 countries.add("India");  
4 countries.add("Japan");  
5 countries.add("Switzerland");  
6 countries.add("Japan");  
7 countries.add("Brazil");  
8  
9 System.out.println(countries); /* [Japan, Brazil, Switzerland, India] */  
10 System.out.println(countries.contains("Switzerland")); /* true */
```

TREESET

```
1 SortedSet<Integer> sortedSet = new TreeSet<>();
2
3 sortedSet.add(10);
4 sortedSet.add(15);
5 sortedSet.add(13);
6 sortedSet.add(21);
7 sortedSet.add(17);
8
9 System.out.println(sortedSet); /* [10, 13, 15, 17, 21] */
10
11 System.out.println(sortedSet.headSet(15)); /* [10, 13] */
12 System.out.println(sortedSet.tailSet(15)); /* [15, 17, 21] */
13
14 System.out.println(sortedSet.subSet(13,17)); /* [13, 15] */
15
16 System.out.println(sortedSet.first()); /* minimum is 10 */
17 System.out.println(sortedSet.last()); /* maximum is 21 */
```

LINKEDHASHSET

```
1 Set<Character> characters = new LinkedHashSet<>();
2
3 for (char c = 'a'; c <= 'k'; c++) {
4     characters.add(c);
5 }
6
7 System.out.println(characters); // [a, b, c, d, e, f, g, h, i, j, k]
```

SET OPERATIONS

```
1 /* getting a mutable set from an immutable one */
2 Set<String> countries = new HashSet<>(List.of("India", "Japan", "Switzerland"));
3
4 countries.addAll(List.of("India", "Germany", "Algeria"));
5 System.out.println(countries); /* [Japan, Algeria, Switzerland, Germany, India] */
6
7 countriesretainAll(List.of("Italy", "Japan", "India", "Germany"));
8 System.out.println(countries); /* [Japan, Germany, India] */
9
10 countries.removeAll(List.of("Japan", "Germany", "USA"));
11 System.out.println(countries); /* [India] */
```

```
1 Set<String> countries = new HashSet<>(List.of("India", "Japan", "Algeria"));
2
3 System.out.println(countries.containsAll(Set.of())); /* true */
4 System.out.println(countries.containsAll(Set.of("India", "Japan"))); /* true */
5 System.out.println(countries.containsAll(Set.of("India", "Germany"))); /* false */
6 System.out.println(countries.containsAll(Set.of("Algeria", "India", "Japan"))); /* true */
```

SET EQUALITY

```
1 Objects.equals(Set.of(1, 2, 3), Set.of(1, 3));      /* false */
2 Objects.equals(Set.of(1, 2, 3), Set.of(1, 2, 3)); /* true */
3 Objects.equals(Set.of(1, 2, 3), Set.of(1, 3, 2)); /* true */
4
5 Set<Integer> numbers = new HashSet<>();
6
7 numbers.add(1);
8 numbers.add(2);
9 numbers.add(3);
10
11 Objects.equals(numbers, Set.of(1, 2, 3)); // true
```

THE MAP INTERFACE

```
Keys  : Values
-----
Bob   : +1-202-555-0118
James : +1-202-555-0220
Katy  : +1-202-555-0175
```

IMMUTABLE MAPS

```
1 Map<String, String> emptyMap = Map.of();  
2  
3 Map<String, String> friendPhones = Map.of(  
4     "Bob", "+1-202-555-0118",  
5     "James", "+1-202-555-0220",  
6     "Katy", "+1-202-555-0175"  
7 );
```

```
1 System.out.println(emptyMap.size());      /* 0 */  
2 System.out.println(friendPhones.size()); /* 3 */
```

```
1 String bobPhone = friendPhones.get("Bob"); /* +1-202-555-0118 */  
2 String alicePhone = friendPhones.get("Alice"); /* null */  
3 String phone = friendPhones.getOrDefault("Alex", "Unknown phone"); /* Unknown phone */
```

```
1 System.out.println(friendPhones.keySet()); /* [James, Bob, Katy] */  
2 System.out.println(friendPhones.values()); /* [+1-202-555-0220, +1-202-555-0118, +1-202-
```

HASHMAP

- represents a map backed by a hash table

```
1 Map<Integer, String> products = new HashMap<>();
2 products.put(1000, "Notebook");
3 products.put(2000, "Phone");
4 products.put(3000, "Keyboard");
5
6 System.out.println(products); /* {2000=Phone, 1000=Notebook, 3000=Keyboard} */
7 System.out.println(products.get(1000)); /* Notebook */
8
9 products.remove(1000);
10 System.out.println(products.get(1000)); /* null */
11 products.putIfAbsent(3000, "Mouse"); /* it does not change the current element */
12 System.out.println(products.get(3000)); /* Keyboard */
```

ESSENTIAL STANDARD CLASSES

LOCALDATE CLASS

- Represents a date (YYYY-MM-dd)
- Part of `java.time` package
- Immutable class
- All operations create new instances

CREATING LOCALDATE INSTANCES

```
1 import java.time.LocalDate;
2
3 /* Get current date */
4 LocalDate now = LocalDate.now();
5
6 /* Create specific dates */
7 LocalDate date1 = LocalDate.of(2017, 11, 25);          /* Using of() */
8 LocalDate date2 = LocalDate.parse("2017-11-25");      /* Using parse() */
9
10 /* Create using day of year */
11 LocalDate date3 = LocalDate.ofYearDay(2017, 33);      /* 2017-02-02 */
12
13 /* Be careful with leap years */
14 LocalDate leapDay = LocalDate.ofYearDay(2016, 366); /* OK - leap year */
15 /* LocalDate error = LocalDate.ofYearDay(2017, 366); */ /* Exception! */
```

GETTING DATE INFORMATION

```
1 LocalDate date = LocalDate.of(2017, 11, 25);
2
3 /* Get basic components */
4 int year = date.getYear();          /* 2017 */
5 int month = date.getMonthValue();   /* 11 */
6 int dayOfMonth = date.getDayOfMonth(); /* 25 */
7 int dayOfYear = date.getDayOfYear(); /* 329 */
8
9 /* Get length information */
10 int lenOfYear = date.lengthOfYear(); /* 365 */
11 int lenOfMonth = date.lengthOfMonth(); /* 30 */
12
13 /* Creating example date */
14 LocalDate newYear = LocalDate.of(2017, 1, 1);
15
16 /* Getting day of week */
17 DayOfWeek dayOfWeek = newYear.getDayOfWeek();
```

DATE ARITHMETIC

```
1 /* Start with January 1, 2017 */
2 LocalDate date = LocalDate.of(2017, 1, 1);
3
4 /* Adding days/months/years */
5 LocalDate tomorrow = date.plusDays(1);      /* 2017-01-02 */
6 LocalDate nextMonth = date.plusMonths(1);    /* 2017-02-01 */
7 LocalDate nextYear = date.plusYears(1);       /* 2018-01-01 */
8
9 /* Subtracting days/months/years */
10 LocalDate yesterday = date.minusDays(1);     /* 2016-12-31 */
11 LocalDate lastMonth = date.minusMonths(1);   /* 2016-12-01 */
12 LocalDate lastYear = date.minusYears(1);      /* 2016-01-01 */
13
14 /* Changing specific components */
15 LocalDate newYear = date.withYear(2016);     /* 2016-01-01 */
16 LocalDate newMonth = date.withMonth(12);      /* 2017-12-01 */
17 LocalDate newDay = date.withDayOfMonth(15);   /* 2017-01-15 */
```

THE INSTANT CLASS

- Represents a moment in time (timestamp)
- Part of `java.time` package
- Stores seconds since epoch (1970-01-01T00:00:00Z)
- Suitable for event timestamps

CREATING INSTANT OBJECTS

```
1 /* Get epoch (1970-01-01T00:00:00Z) */
2 Instant epoch = Instant.EPOCH;
3 /* Creating from epoch values */
4 long posValue = 1234567890L;
5 Instant milli = Instant.ofEpochMilli(posValue);      /* milliseconds */
6 Instant second = Instant.ofEpochSecond(posValue);    /* seconds */
7 Instant secondNano = Instant.ofEpochSecond(posValue, 123L); /* with nanos */
8 /* Using parse method */
9 Instant parsed = Instant.parse("2009-02-14T03:31:30Z");
10 /* With time zones */
11 Instant instant = Instant.ofEpochSecond(1234567890L);
12 System.out.println(instant);                         /* GMT0 */
13 System.out.println(instant.atZone(ZoneId.of("GMT+4"))); /* GMT+4 */
14 System.out.println(instant.atZone(ZoneId.systemDefault())); /* System zone */
```

INSTANT OPERATIONS

```
1 /* Comparing instants */
2 Instant instant1 = Instant.ofEpochSecond(123456L);
3 Instant instant2 = Instant.ofEpochSecond(123456789L);
4 System.out.println(instant1.isAfter(instant2)); /* false */
5 System.out.println(instant1.isBefore(instant2)); /* true */
6 System.out.println(instant1.compareTo(instant2)); /* -1 */
7 /* Adding and subtracting time */
8 Instant instant = Instant.ofEpochSecond(123456L);
9 System.out.println(instant.minus(Duration.ofDays(1)));
10 System.out.println(instant.plus(1, ChronoUnit.DAYS));
11 /* Getting time difference */
12 long days = Instant.EPOCH.until(instant, ChronoUnit.DAYS);
13 long hours = Instant.EPOCH.until(instant, ChronoUnit.HOURS);
```

GET OPERATIONS AND LOCALDATETIME COMPARISON

```
1 /* Getting components */
2 Instant inst = Instant.ofEpochSecond(123456L, 789L);
3 System.out.println(inst.getEpochSecond());           /* seconds */
4 System.out.println(inst.get(ChronoField.NANO_OF_SECOND)); /* nanos */
5
6 /* Differences from LocalDateTime */
7 Instant instant = Instant.now();      /* GMT0 timestamp */
8 LocalDateTime dateTime = LocalDateTime.now(); /* System time */
9
10 /* Getting specific fields */
11 long seconds = inst.getLong(ChronoField.INSTANT_SECONDS);
12 int nanos = inst.get(ChronoField.NANO_OF_SECOND);
13
14 /* Time zone conversion */
15 ZonedDateTime zoned = instant.atZone(ZoneId.systemDefault());
```

BIGDECIMAL CLASS

- For extremely large decimal numbers
- Provides precise decimal arithmetic
- Part of `java.math` package
- Immutable class
- Limited only by available memory

CREATING BIGDECIMAL OBJECTS

```
1 /* Import the class */
2 import java.math.BigDecimal;
3
4 /* Create from String (recommended) */
5 BigDecimal number = new BigDecimal("1000000000000.5897654329");
6
7 /* Using constants */
8 BigDecimal zero = BigDecimal.ZERO;          /* 0 */
9 BigDecimal one = BigDecimal.ONE;           /* 1 */
10 BigDecimal ten = BigDecimal.TEN;           /* 10 */
11
12 /* Comparing with double precision issues */
13 System.out.println(0.1 + 0.2);             /* 0.3000000000000004 */
14 BigDecimal precise = new BigDecimal("0.1")
15     .add(new BigDecimal("0.2"));           /* exactly 0.3 */
```

ARITHMETIC OPERATIONS 1

```
1 /* Basic operations */
2 BigDecimal first = new BigDecimal("0.2");
3 BigDecimal second = new BigDecimal("0.1");
4
5 BigDecimal addition = first.add(second);          /* 0.3 */
6 BigDecimal subtraction = first.subtract(second);   /* 0.1 */
7 BigDecimal multiplication = first.multiply(second);/* 0.02 */
8 BigDecimal division = first.divide(second);        /* 2 */
9 BigDecimal remainder = first.remainder(second);    /* 0.0 */
```

ARITHMETIC OPERATIONS 2

```
1 /* Unary operations */
2 BigDecimal absolute = first.abs();          /* absolute value */
3 BigDecimal power = first.pow(3);            /* raise to power */
4
5 /* Division with scale */
6 BigDecimal dividend = new BigDecimal("1");
7 BigDecimal divisor = new BigDecimal("3");
8 BigDecimal quotient = dividend.divide(divisor, 2, RoundingMode.HALF_EVEN); /* 0.33 */
```

ROUNDING AND SCALE 1

```
1 /* Import rounding mode */
2 import java.math.RoundingMode;
3
4 /* Getting current scale */
5 BigDecimal number = new BigDecimal("123.4567");
6 System.out.println(number.scale()); /* 4 */
7
8 /* Setting scale with rounding */
9 BigDecimal rounded = number.setScale(2, RoundingMode.HALF_UP); /* 123.46 */
```

ROUNDING AND SCALE 2

```
1 /* Different rounding modes */
2 BigDecimal value = new BigDecimal("100.5649");
3 System.out.println(value.setScale(3, RoundingMode.CEILING));      /* 100.565 */
4 System.out.println(value.setScale(1, RoundingMode.HALF_DOWN));    /* 100.6 */
5
6 /* Remember immutability */
7 BigDecimal bd = new BigDecimal("999.999");
8 bd.setScale(2, RoundingMode.HALF_UP);      /* Original unchanged */
9 bd = bd.setScale(2, RoundingMode.HALF_UP); /* Assigns new value */
```

LAMBDA EXPRESSIONS

LAMBDA EXPRESSION

```
1 (parameter list) -> { body of the lambda expression }
```

```
1 BiFunction<Integer, Integer, Boolean> isDivisible = (x, y) -> x % y == 0;
```

```
1 /* if it has only one argument "()" are optional */
2 Function<Integer, Integer> adder1 = x -> x + 1;
3
4 /* with type inference */
5 Function<Integer, Integer> mult2 = (Integer x) -> x * 2;
6
7 /* with multiple statements */
8 Function<Integer, Integer> adder5 = (x) -> {
9     x += 2;
10    x += 3;
11    return x;
12 };
```

INVOKING LAMBDA EXPRESSIONS

```
1 BiFunction<Integer, Integer, Boolean> isDivisible = (x, y) -> x % y == 0;  
2 boolean result4Div2 = isDivisible.apply(4, 2); /* true */  
3 boolean result3Div5 = isDivisible.apply(3, 5); /* false */
```

PASSING LAMBDA EXPRESSIONS TO METHODS

```
1 private static void printResultOfLambda(Function<String, Integer> function) {  
2     System.out.println(function.apply("HAPPY NEW YEAR 3000!"));  
3 }
```

```
1 /* it returns the length of a string */  
2 Function<String, Integer> f = s -> s.length();  
3 printResultOfLambda(f); /* it prints 20 */
```

PASSING LAMBDA EXPRESSIONS TO METHODS

```
1 /* It prints the number of digits: 4 */
2 printResultOfLambda(s -> {
3     int count = 0;
4     for (char c : s.toCharArray()) {
5         if (Character.isDigit(c)) {
6             count++;
7         }
8     }
9     return count;
10});
```

CLOSURES

```
1 final String hello = "Hello, ";
2 Function<String, String> helloFunction = (name) -> hello + name;
3
4 System.out.println(helloFunction.apply("John"));
5 System.out.println(helloFunction.apply("Anastasia"));
```

```
1 Hello, John
2 Hello, Anastasia
```

```
1 int constant = 100;
2 Function<Integer, Integer> adder100 = x -> x + constant;
3
4 System.out.println(adder100.apply(200)); /* 300 */
5 System.out.println(adder100.apply(300)); /* 400 */
```

FUNCTIONAL INTERFACES

```
1 @FunctionalInterface  
2 interface Func<T, R> {  
3     R apply(T val); /* single abstract method */  
4 }
```

FUNCTIONAL INTERFACES

```
1 @FunctionalInterface
2 interface Func<T, R> {
3     R apply(T val);
4
5     static void doNothingStatic() { }
6
7     default void doNothingByDefault() { }
8 }
```

IMPLEMENTING FUNCTIONAL INTERFACES

1. Anonymous classes

```
1 Func<Long, Long> square = new Func<Long, Long>() {
2     @Override
3     public Long apply(Long val) {
4         return val * val;
5     }
6 };
7
8 long val = square.apply(10L); /* the result is 100L */
```

IMPLEMENTING FUNCTIONAL INTERFACES

2.Lambda expressions

```
1 Func<Long, Long> square = val -> val * val; /* the lambda expression */
2
3 long val = square.apply(10L); /* the result is 100L */
```

IMPLEMENTING FUNCTIONAL INTERFACES

3.Method references

```
1 class Functions {  
2  
3     public static long square(long val) {  
4         return val * val;  
5     }  
6 }
```

```
1 Func<Long, Long> square = Functions::square;
```

METHOD REFERENCES

```
1 objectOrClass :: methodName
```

```
1 BiFunction<Integer, Integer, Integer> max = Integer::max;
```

```
1 System.out.println(max.apply(50, 70)); // 70
```

```
1 BiFunction<Integer, Integer, Integer> max = (x, y) -> Integer.max(x, y);
```

KINDS OF METHOD REFERENCES

1. Reference to a static method

```
1 ClassName :: staticMethodName
```

```
1 Function<Double, Double> sqrt = Math::sqrt;
```

```
1 sqrt.apply(100.0d); // the result is 10.0d
```

```
1 Function<Double, Double> sqrt = x -> Math.sqrt(x);
```

KINDS OF METHOD REFERENCES

2. Reference to an instance method of an object

```
1  objectName :: instanceMethodName
```

```
1  String whatsGoingOnText = "What's going on here?";  
2  
3  Function<String, Integer> indexWithinWhatsGoingOnText = whatsGoingOnText::indexOf;
```

```
1  System.out.println(indexWithinWhatsGoingOnText.apply("going")); /* 7 */  
2  System.out.println(indexWithinWhatsGoingOnText.apply("Hi")); /* -1 */
```

```
1  Function<String, Integer> indexWithinWhatsGoingOnText = string -> whatsGoingOnText.indexOf(string);
```

KINDS OF METHOD REFERENCES

3. Reference to an instance method of an object of a particular type

```
1 ClassName :: instanceMethodName
```

```
1 Function<Long, Double> converter = Long::doubleValue;
```

```
1 converter.apply(100L); /* the result is 100.0d */
2 converter.apply(200L); /* the result is 200.0d */
```

```
1 Function<Long, Double> converter = val -> val.doubleValue();
```

KINDS OF METHOD REFERENCES

4. Reference to a constructor

```
1 ClassName :: new
```

```
1 class Person {
2     String name;
3
4     public Person(String name) {
5         this.name = name;
6     }
7 }
```

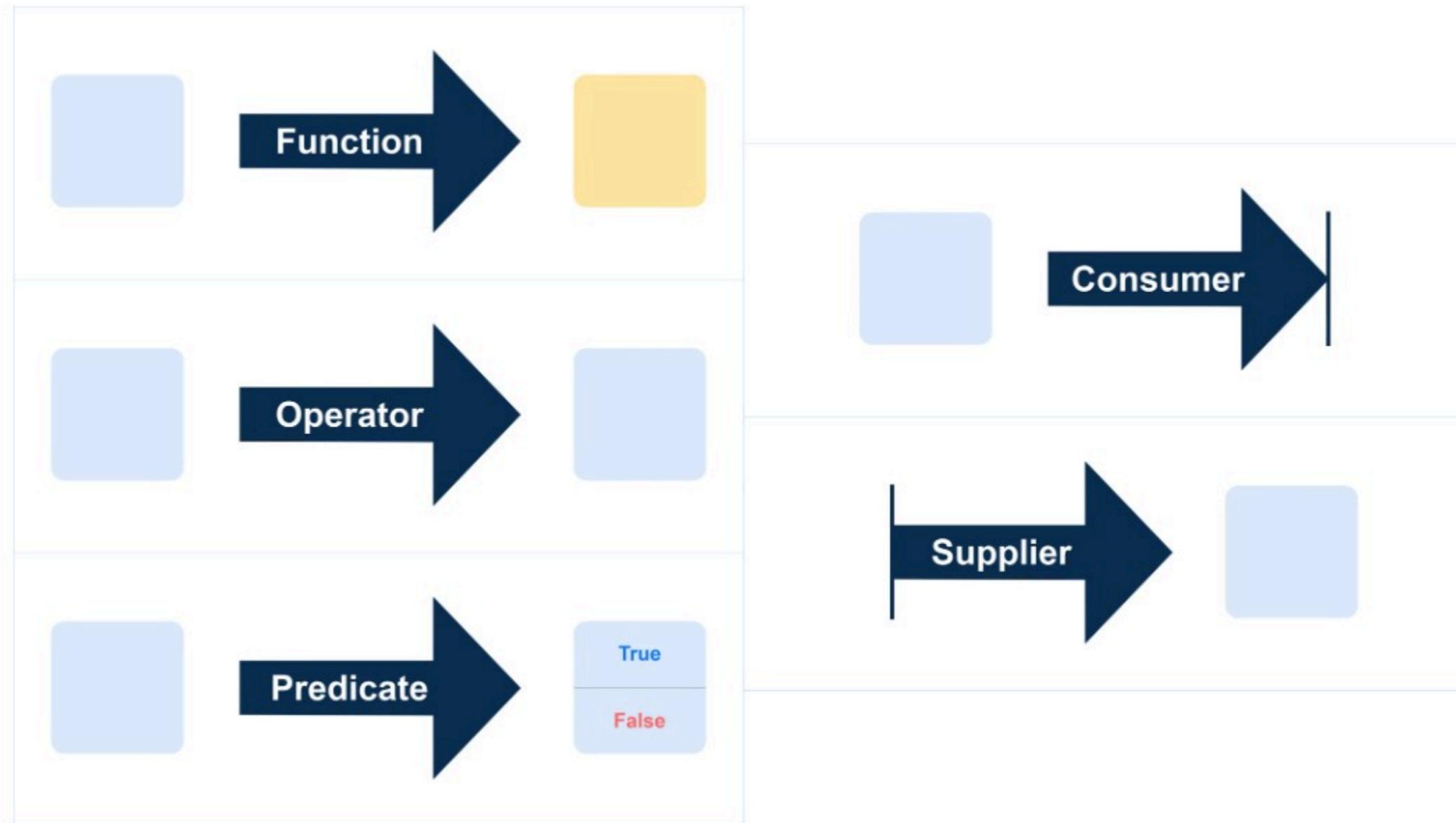
```
1 Function<String, Person> personGenerator = Person::new;
```

```
1 Person johnFoster = personGenerator.apply("John Foster"); // we have a John Foster object
```

```
1 Function<String, Person> personGenerator = name -> new Person(name);
```

FUNCTIONS

FUNCTIONAL INTERFACES



STANDARD FUNCTIONAL INTERFACES WITH EXAMPLES

Functions

```
1 /* String to Integer function */
2 Function<String, Integer> converter = Integer::parseInt;
3 converter.apply("1000"); /* the result is 1000 (Integer) */
4
5 /* String to int function */
6ToIntFunction<String> anotherConverter = Integer::parseInt;
7 anotherConverter.applyAsInt("2000"); /* the result is 2000 (int) */
8
9 /* (Integer, Integer) to Integer function */
10 BiFunction<Integer, Integer, Integer> sumFunction = (a, b) -> a + b;
11 sumFunction.apply(2, 3); /* it returns 5 (Integer) */
```

STANDARD FUNCTIONAL INTERFACES WITH EXAMPLES

Operators

```
1 /* Long to Long multiplier */
2 UnaryOperator<Long> longMultiplier = val -> 100_000 * val;
3 longMultiplier.apply(2L); /* the result is 200_000L (Long) */
4
5 /* int to int operator */
6 IntUnaryOperator intMultiplier = val -> 100 * val;
7 intMultiplier.applyAsInt(10); /* the result is 1000 (int) */
8
9 /* (String, String) to String operator */
10 BinaryOperator<String> appender = (str1, str2) -> str1 + str2;
11 appender.apply("str1", "str2"); /* the result is "str1str2" */
```

STANDARD FUNCTIONAL INTERFACES WITH EXAMPLES

Predicates

```
1 /* Character to boolean predicate */
2 Predicate<Character> isDigit = Character::isDigit;
3 isDigit.test('h'); /* the result is false (boolean) */
4
5 /* int to boolean predicate */
6 IntPredicate isEven = val -> val % 2 == 0;
7 isEven.test(10); /* the result is true (boolean) */
```

STANDARD FUNCTIONAL INTERFACES WITH EXAMPLES

Suppliers

```
1 Supplier<String> stringSupplier = () -> "Hello";
2 stringSupplier.get(); /* the result is "Hello" (String) */
3
4 BooleanSupplier booleanSupplier = () -> true;
5 booleanSupplier.getAsBoolean(); /* the result is true (boolean) */
6
7 IntSupplier intSupplier = () -> 33;
8 intSupplier.getAsInt(); /* the result is 33 (int) */
```

STANDARD FUNCTIONAL INTERFACES WITH EXAMPLES

Consumers

```
1 /* it prints a given string */
2 Consumer<String> printer = System.out::println;
3 printer.accept("!!!"); /* It prints "!!!" */
```

FUNCTION COMPOSITION

Two default methods:

- `f.compose(g).apply(x)` is the same as `f(g(x))`
- `f.andThen(g).apply(x)` is the same as `g(f(x))`

```
1 Function<Integer, Integer> adder = x -> x + 10;
2 Function<Integer, Integer> multiplier = x -> x * 5;
3
4 /* compose: adder(multiplier(5)) */
5 System.out.println("result: " + adder.compose(multiplier).apply(5));
6
7 /* andThen: multiplier(add(5)) */
8 System.out.println("result: " + adder.andThen(multiplier).apply(5));
```

```
1 result: 35
2 result: 75
```

COMPOSING PREDICATES

```
1 IntPredicate isOdd = n -> n % 2 != 0; /* it's true for 1, 3, 5, 7, 9, 11 and so on */
2
3 System.out.println(isOdd.test(10)); /* prints "false" */
4 System.out.println(isOdd.test(11)); /* prints "true" */
5
6 IntPredicate lessThan11 = n -> n < 11; /* it's true for all numbers < 11 */
7
8 System.out.println(lessThan11.test(10)); /* prints "true" */
9 System.out.println(lessThan11.test(11)); /* prints "false" */
```

```
1 IntPredicate isEven = isOdd.negate(); /* it's true for 0, 2, 4, 6, 8, 10 and so on */
2 System.out.println(isEven.test(10)); /* prints "true" */
3 System.out.println(isEven.test(11)); /* prints "false" */
```

COMPOSING PREDICATES

```
1 IntPredicate isOddOrLessThan11 = isOdd.or(lessThan11);
2
3 System.out.println(isOddOrLessThan11.test(10)); /* prints "true" */
4 System.out.println(isOddOrLessThan11.test(11)); /* prints "true" */
5 System.out.println(isOddOrLessThan11.test(12)); /* prints "false" */
6 System.out.println(isOddOrLessThan11.test(13)); /* prints "true" */
7
8 IntPredicate isOddAndLessThan11 = isOdd.and(lessThan11);
9
10 System.out.println(isOddAndLessThan11.test(8)); /* prints "false" */
11 System.out.println(isOddAndLessThan11.test(9)); /* prints "true" */
12 System.out.println(isOddAndLessThan11.test(10)); /* prints "false" */
13 System.out.println(isOddAndLessThan11.test(11)); /* prints "false" */
```

COMPOSING CONSUMERS

```
1 Consumer<String> consumer = System.out::println;  
2 Consumer<String> doubleConsumer = consumer.andThen(System.out::println);  
3 doubleConsumer.accept("Hi!");
```

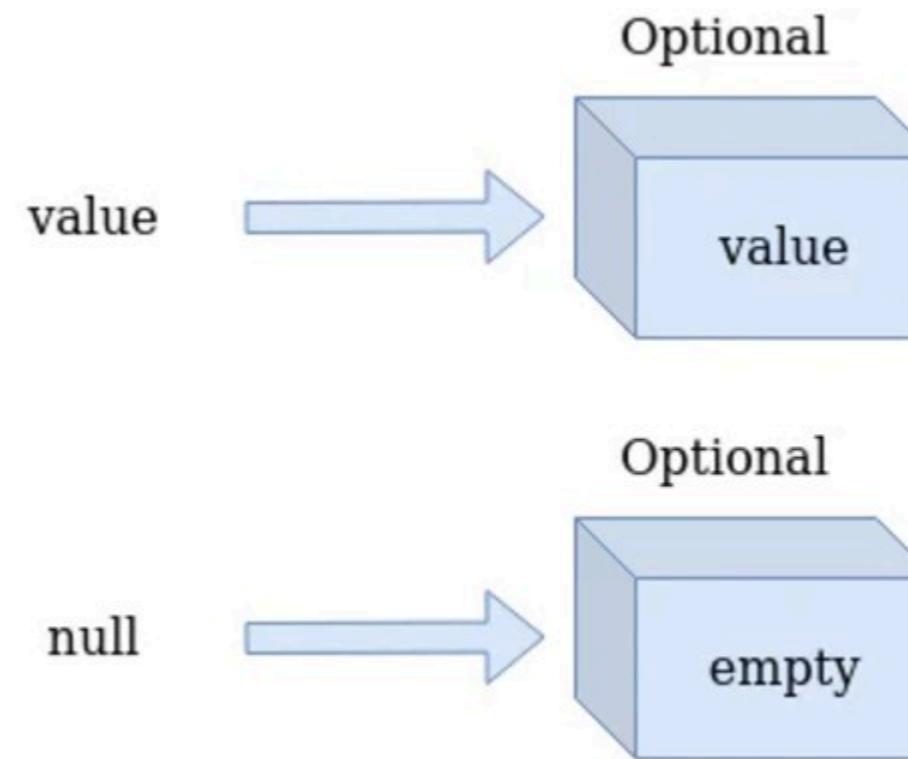
```
1 Hi!  
2 Hi!
```

OPTIONAL

```
1 Optional<String> absent = Optional.empty();  
2 Optional<String> present = Optional.of("Hello");
```

```
1 System.out.println(absent.isPresent()); /* false */  
2 System.out.println(present.isPresent()); /* true */
```

OPTIONALS AND NULLABLE OBJECTS



```
1 String message = getRandomMessage(); /* it may be null */
2
3 Optional<String> optMessage = Optional.ofNullable(message);
4
5 System.out.println(optMessage.isPresent()); /* true or false */
```

GETTING THE VALUE FROM AN OPTIONAL

get - returns the value of the instance if it's present, otherwise throws an exception

```
1 Optional<String> optName = Optional.of("John");  
2 String name = optName.get(); // "John"
```

```
1 Optional<String> optName = Optional.ofNullable(null);  
2 String name = optName.get(); // throws NoSuchElementException
```

GETTING THE VALUE FROM AN OPTIONAL

- `orElse` - used to extract the value wrapped inside an Optional object or return some default value when the Optional is empty

```
1 String nullableName = null;
2 String name = Optional.ofNullable(nullableName).orElse("unknown");
3
4 System.out.println(name); // unknown
```

- `orElseGet` - takes a supplier function to produce a result instead of taking some value to return

```
1 String name = Optional
2     .ofNullable(nullableName)
3     .orElseGet(SomeClass::getDefaultResult);
```

CONDITIONAL ACTIONS

- `ifPresent` -performs the given action with the value, otherwise does nothing
- `ifPresentOrElse` -performs the given action with the value, otherwise performs the given empty-based action

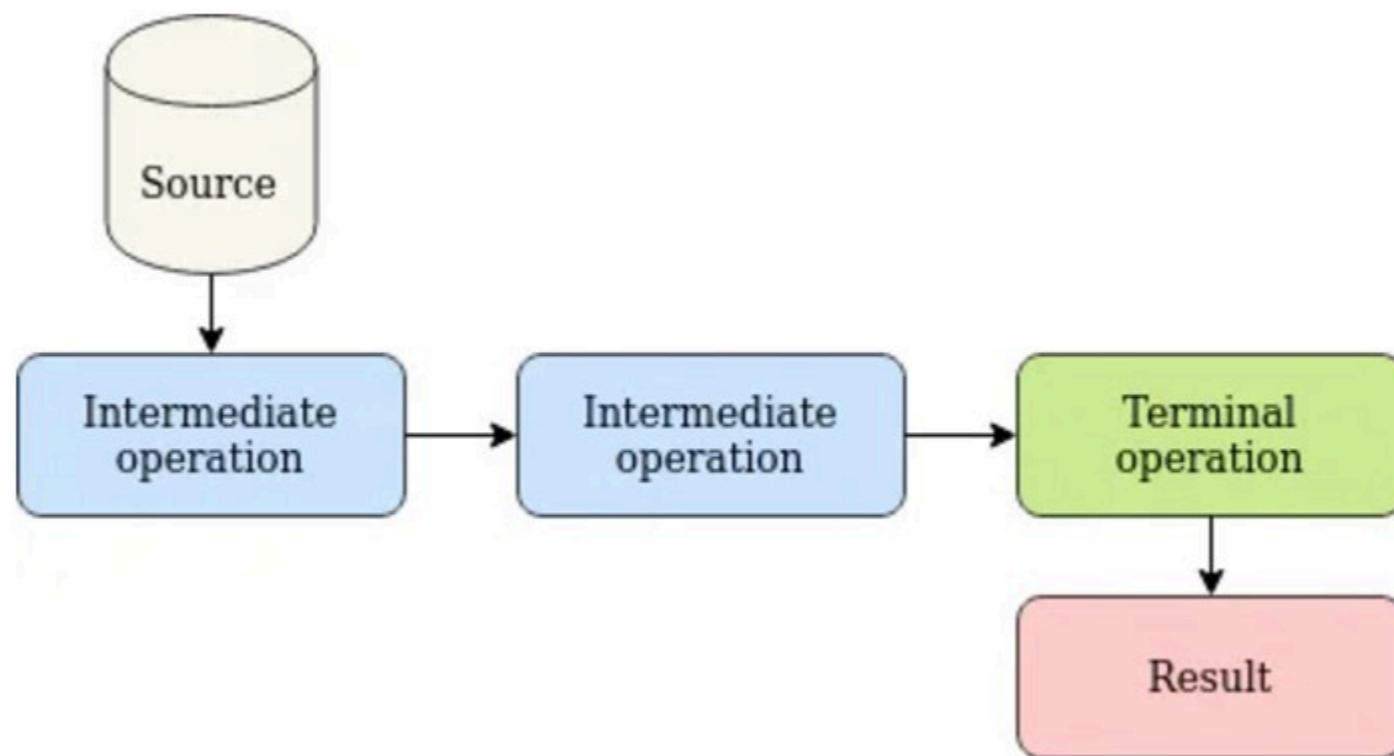
```
1 Optional<String> companyName = Optional.of("Google");
2 companyName.ifPresent((name) -> System.out.println(name.length())); // 6
```

```
1 Optional<String> noName = Optional.empty();
2 noName.ifPresent((name) -> System.out.println(name.length()));
```

```
1 Optional<String> optName = Optional.ofNullable(/* some value goes here */);
2
3 optName.ifPresentOrElse(
4     (name) -> System.out.println(name.length()),
5     () -> System.out.println(0)
6 );
```

FUNCTIONAL STREAMS

FUNCTIONAL DATA PROCESSING



A LOOP VS. A STREAM EXAMPLE

```
1 List<Integer> numbers = List.of(1, 4, 7, 6, 2, 9, 7, 8);
```

```
1 long count = 0;
2 for (int number : numbers) {
3     if (number > 5) {
4         count++;
5     }
6 }
7 System.out.println(count); /* 5 */
```

```
1 long count = numbers.stream()
2         .filter(number -> number > 5)
3         .count(); /* 5 */
```

```
1 long count = numbers.stream()
2         .skip(4) /* skip 1, 4, 7, 6 */
3         .filter(number -> number > 5)
4         .count(); /* 3 */
```

CREATING STREAMS

- from a collection:

```
1 List<Integer> famousNumbers = List.of(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55);
2 Stream<Integer> numbersStream = famousNumbers.stream();
3
4 Set<String> usefulConcepts = Set.of("functions", "lazy", "immutability");
5 Stream<String> conceptsStream = usefulConcepts.stream();
```

- from an array:

```
1 Stream<Double> doubleStream = Arrays.stream(new Double[]{ 1.01, 1d, 0.99, 1.02, 1d, 0.99 })
```

- directly from some values:

```
1 Stream<String> persons = Stream.of("John", "Demetra", "Cleopatra");
```

CREATING STREAMS

- concatenate streams together:

```
1 Stream<String> stream1 = Stream.of(/* some values */);
2 Stream<String> stream2 = Stream.of(/* some values */);
3 Stream<String> resultStream = Stream.concat(stream1, stream2);
```

- empty streams:

```
1 Stream<Integer> empty1 = Stream.of();
2 Stream<Integer> empty2 = Stream.empty();
```

INTERMEDIATE OPERATIONS

- `filter` returns a new stream that includes the elements that match a **predicate**;
- `limit` returns a new stream that consists of the first `n` elements of this stream;
- `skip` returns a new stream without the first `n` elements of this stream;
- `distinct` returns a new stream consisting of only unique elements according to the results of `equals`;
- `sorted` returns a new stream that includes elements sorted according to the natural order or a given **comparator**;
- `peek` returns the same stream of elements but allows observing the current elements of the stream for debugging;
- `map` returns a new stream that consists of the elements that were obtained by applying a function (i.e. transforming each element).

TERMINAL OPERATIONS

- `count` returns the number of elements in the stream as a `long` value;
- `max` / `min` returns an `Optional` maximum/minimum element of the stream according to the given comparator;
- `reduce` combines values from the stream into a single value (an aggregate value);
- `findFirst` / `findAny` returns the first / any element of the stream as an `Optional`;
- `anyMatch` returns `true` if at least one element matches a predicate (see also:
`allMatch` , `noneMatch`);
- `forEach` takes a **consumer** and applies it to each element of the stream (for example, printing it);
- `collect` returns a collection of the values in the stream;
- `toArray` returns an array of the values in a stream.

AN EXAMPLE

```
1 List<String> companies = List.of(  
2     "Google", "Amazon", "Samsung",  
3     "GOOGLE", "amazon", "Oracle"  
4 );  
5  
6 companies.stream()  
7     .map(String::toUpperCase) /* transform each name to the upper case*/  
8     .distinct() /* intermediate operation: keep only unique words*/  
9     .forEach(System.out::println); /* print every company*/
```

```
1 GOOGLE  
2 AMAZON  
3 SAMSUNG  
4 ORACLE
```

THE MAP OPERATION

```
1 List<Double> numbers = List.of(6.28, 5.42, 84.0, 26.0);
```

```
1 List<Double> famousNumbers = numbers.stream()  
2     .map(number -> number / 2) /* divide each number in the stream by 2 */  
3     .collect(Collectors.toList()); /* collect transformed numbers in a new list */
```

```
1 [3.14, 2.71, 42.0, 13.0]
```

THE MAP OPERATION

```
1 public class Job {  
2     private String title;  
3     private String description;  
4     private double salary;  
5  
6     /* getters and setters */  
7 }
```

```
1 List<String> titles = jobs.stream()  
2         .map(Job::getTitle) /* get title of each job */  
3         .collect(Collectors.toList()); /* collect titles in a new list */
```

THE MAP OPERATION

```
1 class User {  
2     private long id;  
3     private String firstName;  
4     private String lastName;  
5 }  
6  
7 class Account {  
8     private long id;  
9     private boolean isLocked;  
10    private User owner;  
11 }  
12  
13 class AccountInfo {  
14     private long id;  
15     private String ownerFullName;  
16 }
```

```
1 List<AccountInfo> infoList = accounts.stream()  
2     .map(acc -> {  
3         AccountInfo info = new AccountInfo();  
4         info.setId(acc.getId());  
5         String ownerFirstName = acc.getOwner().getFirstName();  
6         String ownerLastName = acc.getOwner().getLastName();  
7         info.setOwnerFullName(ownerFirstName + " " + ownerLastName);  
8         return info;  
9     }).collect(Collectors.toList());
```

PRIMITIVE-SPECIALIZED TYPES OF THE MAP OPERATION

```
1 class Planet {  
2     private String name;  
3     private int orderFromSun;  
4  
5     public Planet(int orderFromSun) {  
6         this.orderFromSun = orderFromSun;  
7     }  
8 }
```

```
1 List<Planet> planets = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8)  
2         .mapToObj(Planet::new)  
3         .collect(Collectors.toList());
```

THE FLATMAP OPERATION

```
1 List<Book> javaBooks = List.of(  
2     new Book("Java EE 7 Essentials", 2013, List.of("Arun Gupta")),  
3     new Book("Algorithms", 2011, List.of("Robert Sedgewick", "Kevin Wayne")),  
4     new Book("Clean code", 2014, List.of("Robert Martin"))  
5 );
```

```
1 List<String> authors = javaBooks.stream()  
2     .flatMap(book -> book.getAuthors().stream())  
3     .collect(Collectors.toList());
```

```
1 ["Arun Gupta", "Robert Sedgewick", "Kevin Wayne", "Robert Martin"]
```

THE REDUCE OPERATION

- reduce is a method of a Stream class that combines elements of a stream into a single value. The result can be a value of a primitive type or a complex object.

```
1 List<Integer> transactions = List.of(20, 40, -60, 5);  
2 transactions.stream().reduce((sum, transaction) -> sum + transaction);
```

iteration	sum	transaction
0	20	40
1	20 + 40 = 60	-60
2	60 + -60 = 0	5
return	0 + 5 = 5	

OTHER REDUCTION OPERATIONS

```
1 transactions.stream().reduce((t1, t2) -> t2 > t1 ? t2 : t1)
```

```
1 transactions.stream().max(Integer::compareTo);
```

```
1 IntStream.of(20, 40, -60, 5).max();
```

STREAM FILTERING

```
1 List<Integer> primeNumbers = Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);  
  
1 List<Integer> filteredPrimeNumbers = primeNumbers.stream() /* create a stream from the l  
2     .filter(n -> n >= 11 && n <= 23) /* filter elements */  
3     .collect(Collectors.toList()); /* collect elements in a new list */  
  
1 [11, 13, 17, 19, 23]  
  
1 Predicate<Integer> between11and23 = n -> n >= 11 && n <= 23; /* instantiate the predicit  
2  
3 List<Integer> filteredPrimeNumbers = primeNumbers.stream() /* create a stream from the l  
4     .filter(between11and23) /* pass the predicate to the filter method */  
5     .collect(Collectors.toList()); /* collect elements in a new list */
```

USING MULTIPLE FILTERS

```
1 List<String> programmingLanguages = Arrays.asList("Java", "", "scala", "Kotlin", "", "cl
```

```
1 long count = programmingLanguages.stream()
  2     .filter(lang -> lang.length() > 0) /* consider only non-empty strings */
  3     .filter(lang -> Character.isUpperCase(lang.charAt(0)))
  4     .count(); /* count suitable languages */
```

```
1 filter(lang -> lang.length() > 0 && Character.isUpperCase(lang.charAt(0)))
```

COLLECTORS

```
1 public class Account {  
2     private long balance;  
3     private String number;  
4  
5     /* getters and setters */  
6 }
```

```
1 List<Account> accounts = accountStream.collect(Collectors.toList());
```

```
1 Set<Account> accounts = accountStream.collect(Collectors.toSet());
```

```
1 LinkedList<Account> accounts = accountStream.collect(Collectors.toCollection(LinkedList::
```

PRODUCING VALUES

- `summingInt`, `summingLong`, `summingDouble`
- `averagingInt`, `averagingLong`, `averagingDouble`
- `maxBy`, `minBy`
- `counting`

```
1 long summary = accounts.stream()
2     .collect(summingLong(Account::getBalance));
```

```
1 double average = accounts.stream()
2     .collect(averagingLong(Account::getBalance));
```

```
1 String megaNumber = accountStream.collect(Collectors.reducing("", 
2     account -> account.getNumber(),
3     (numbers, number) -> numbers.concat(number)
4 ));
```

STREAM PIPELINES

```
1 List<String> words = List.of("JAR", "Java", "Kotlin", "JDK", "jakarta");
2
3 long numberOfWords = words.stream()
4     .map(String::toUpperCase)          /* convert all words to upper case */
5     .filter(s -> s.startsWith("JA"))  /* filter words using a prefix */
6     .count();                         /* count the suitable words */
7
8 System.out.println(numberOfWords); /* 3 */
```



THE ORDER OF EXECUTION

```
1 long numberOfWords = words.stream()
2     .map(String::toUpperCase)
3     .peek(System.out::println)
4     .filter(s -> s.startsWith("JA"))
5     .peek(System.out::println)
6     .count();
```

Output:

```
1 JAR
2 JAR
3 JAVA
4 JAVA
5 KOTLIN
6 JDK
7 JAKARTA
8 JAKARTA
```

STREAMS WITH CUSTOM CLASSES

Let's assume that we have the `Event` class that represents a public event, such as a conference, a film premiere, or a concert. It has two fields:

- `beginning (LocalDate)` is a date when the event happens;
- `name (String)` that is the name of the event (for instance, "JavaOne – 2017").

Also, the class has getters and setters for each field with the corresponding names.

We also have a list of instances named `events`

```
1 LocalDate after = LocalDate.of(2017, 12, 29);
2 LocalDate before = LocalDate.of(2018, 1, 1);
3
4 List<String> suitableEvents = events.stream()
5     .filter(e -> e.getBeginning().isAfter(after) && e.getBeginning().isBefore(before))
6     .map(Event::getName)
7     .collect(Collectors.toList());
```

MAPPING AND REDUCING FUNCTIONS

```
1 public static IntPredicate negateEachAndConjunctAll(Collection<IntPredicate> predicates)
2     return predicates.stream()
3         .map(IntPredicate::negate)
4         .reduce(n -> true, IntPredicate::and);
5 }
```

⇒ the input predicates $P_1(x), P_2(x), \dots, P_n(x)$ will be reduced into one predicate $Q(x) = \text{not } P_1(x) \text{ and not } P_2(x) \text{ and ... and not } P_n(x)$

TAKING ELEMENTS

THE TAKEWHILE METHOD

```
1 List<Integer> numbers =  
2     Stream.of(3, 5, 1, 2, 0, 4, 5)  
3         .takeWhile(n -> n > 0)  
4         .collect(Collectors.toList());  
5  
6 System.out.println(numbers); // [3, 5, 1, 2]
```

THE DROPWHILE METHOD

```
1 List<Integer> numbers =  
2     Stream.of(3, 5, 1, 2, 0, 4, 5)  
3         .dropWhile(n -> n > 0)  
4         .collect(Collectors.toList());  
5  
6 System.out.println(numbers); // [0, 4, 5]
```

THE CASE OF UNORDERED STREAMS

```
1 Set<String> conferences = Set.of(  
2     "JokerConf", "JavaZone",  
3     "KotlinConf", "JFokus"  
4 );  
5  
6 conferences.stream()  
7     .takeWhile(word -> word.startsWith("J"))  
8     .forEach(System.out::println);
```

GROUPING COLLECTORS

PARTITIONING

```
1 List<Account> accounts = List.of(  
2     new Account(3333, "530012"),  
3     new Account(15000, "771843"),  
4     new Account(0, "681891")  
5 );
```

```
1 Map<Boolean, List<Account>> accountsByBalance = accounts.stream()  
2     .collect(Collectors.partitioningBy(account -> account.getBalance() >= 10000));
```

```
1 {  
2     false=[Account{balance=3333, number='530012'}, Account{balance=0, number='681891'}],  
3     true=[Account{balance=15000, number='771843'}]  
4 }
```

GROUPING

```
1 enum Status {  
2     ACTIVE,  
3     BLOCKED,  
4     REMOVED  
5 }  
6  
7 public class Account {  
8     private long balance;  
9     private String number;  
10    private Status status;  
11  
12    /* constructors  
13        getters and setters */  
14 }
```

```
1 List<Account> accounts = List.of(  
2     new Account(3333L, "530012", Status.REMOVED),  
3     new Account(15000L, "771843", Status.ACTIVE),  
4     new Account(0L, "681891", Status.BLOCKED)  
5 );
```

```
1 Map<Status, List<Account>> accountsByStatus = accounts.stream()  
2         .collect(Collectors.groupingBy(Account::getStatus));
```

```
1 {  
2     BLOCKED=[Account{balance=0, number='681891'}],  
3     REMOVED=[Account{balance=3333, number='530012'}],
```

DOWNSTREAM COLLECTORS

```
1 List<Account> accounts = List.of(  
2     new Account(3333L, "530012", Status.ACTIVE),  
3     new Account(15000L, "771843", Status.BLOCKED),  
4     new Account(15000L, "234465", Status.ACTIVE),  
5     new Account(8800L, "110011", Status.ACTIVE),  
6     new Account(45000L, "462181", Status.BLOCKED),  
7     new Account(0L, "681891", Status.REMOVED)  
8 );
```

```
1 Map<Status, Long> sumByStatuses = accounts.stream()  
2     .collect(Collectors.groupingBy(  
3         Account::getStatus,  
4         Collectors.summingLong(Account::getBalance))  
5     );
```

```
1 { REMOVED=0, ACTIVE=27133, BLOCKED=60000 }
```

TEEING COLLECTOR

```
1 long[ ] countAndSum = accounts
2     .stream()
3     .filter(account -> account.getStatus() == Status.BLOCKED)
4     .collect(Collectors.teeing(
5         Collectors.counting(),
6         Collectors.summingLong(Account::getBalance),
7         (count, sum) -> new long[ ]{count, sum}
8     );
```



RESOURCES

- Effective Java 3rd Edition
- Functional Programming in Java, Second Edition
- Oracle Java Tutorials
- Oracle's own Java learning platform
- University of Helsinki Object-Oriented Programming with Java

