# Introduction to Java Programming

Danciu Gabriel Mihail

November 9, 2023

# Contents

# Chapter 1

# Introduction

This chapter provides an introduction to the book and its contents. We will explain what the book is about, who it is intended for, and what the reader can expect to learn from it.

## 1.1 Purpose and Scope of the Book

The purpose of this book is to provide a comprehensive introduction to Java programming for beginners. It covers the basic concepts and principles of Java programming, as well as more advanced topics such as application development and third-party libraries.

The scope of this book is limited to the Java programming language and its related tools and libraries. We will not cover other programming languages or technologies in depth, although we will make occasional comparisons and references to related technologies Schildt (2021).

## 1.2 Intended Audience

This book is intended for anyone who wants to learn Java programming, regardless of their prior programming experience. It is suitable for students, hobbyists, and professionals who want to learn a new programming language or improve their existing skills.

No prior programming experience is required, although familiarity with basic computer concepts such as file systems, data types, and algorithms is helpful.

## 1.3 Book Organization

This book is organized into several chapters, each covering a different aspect of Java programming. The chapters are organized as follows:
- Chapter 1: Introduction
- Chapter 2: Overview of Java
- Chapter 3: Basic Java Programming Concepts
- Chapter 4: Object-Oriented Programming Java

- Chapter 5: Advanced Object-Oriented Programming Concepts
- Chapter 6: Java Collections
- Chapter 7: More Java Programming Concepts
- Chapter 8: A closer look to Java Collections
- Chapter 9: Java Application Development
- Chapter 10: Java Tools and Libraries
- Chapter 11: Laboratory exercises

Each chapter is further divided into sections and subsections, which provide a more detailed explanation of the concepts covered in the chapter.

## 1.4 Why Learn Java?

Java is one of the most popular programming languages in the world, and for good reason. It is used by millions of developers to create a wide variety of applications, from desktop software to mobile apps and web services.

Some of the benefits of learning Java include:
- Job opportunities: Java is one of the most in-demand programming languages in the job market, and is used by many large companies such as Google, Amazon, and Oracle.
- Versatility: Java can be used to create a wide range of applications, from simple command-line tools to complex web applications and mobile apps.
- Robustness: Java's strict type checking and runtime checking help ensure that programs are free from errors and crashes.
- Portability: Java programs can run on different platforms without modification, thanks to the Java Virtual Machine (JVM).
- Security: Java's security model helps protect against malicious code and other security threats.

Overall, learning Java is a valuable skill for anyone interested in software development, and can open up many career opportunities.

# Chapter 2

# Overview of Java

This chapter provides an overview of Java programming language. We will discuss the history of Java, its features and characteristics, and its role in the world of software development.

## 2.1 What is Java?

Java was created in the mid-1990s by a team of developers at Sun Microsystems, led by James Gosling. The goal was to create a programming language that was simple, robust, and portable.

Initially, Java was designed for use in consumer electronics devices such as set-top boxes and handheld devices. However, it quickly gained popularity as a general-purpose programming language, thanks to its unique features and capabilities. Java is one of the most widely used high-level programming languages, but that is not its primary merit. This language has revolutionized programming in many ways, which we will detail in this course. The purpose of this book is to present this language at a medium level and to use it to understand the concepts of data structures.

The Java programming language is the language in which applications, applets, and servlets can be written. When a Java program is compiled, the source code will be converted into byte code, which is a machine language that is portable on any CPU architecture. This is possible due to the existence of a JVM virtual machine, which facilitates the interpretation of byte code into machine language specific to the machine on which the program will be run.

The Java platform is the set of Java classes that exist in any Java installation kit. These classes can be used by any Java application running on the computer where they were installed. The Java platform is also called the Java environment or the Java API kernel (Application Programming Interface). Another name for these sets of classes is a framework.

Java classes are grouped into collections of classes called packages. We will detail their utility later in this course. Packages are also organized according to their role/function, such as network packages, graphics, user interface manipulation, security, etc.

The Java programming language is an object-oriented (OO) language, similar to C++, very powerful and easy to use and especially easy to learn by programmers. It is the result of many years of work and incorporates elegant design and cutting-edge features, making it quite popular among programmers. Versatility, flexibility, efficiency, and portability are other aspects that

propel Java ahead of others.

In addition to these, the fact that programmers can create programs that can run in browsers or web services, or that they can create applications that run on different platforms, or the fact that they can create programs that run on almost any newer electronic device (mobile, medical, industrial, remote, etc.), make this language very powerful.

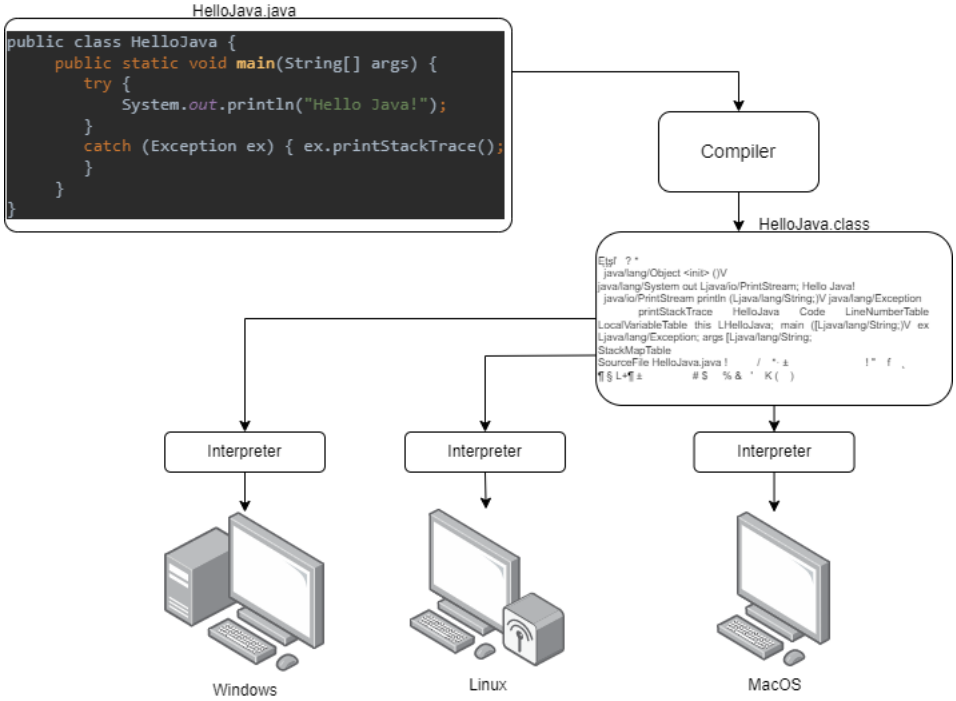



Figure 2.1: Java can run almost anywhere

The Java virtual machine constitutes the fundamental element of Java. Java programs are portable on any operating system, hardware architecture that supports a Java interpreter. Sun, the company that produces various VM kits (Virtual Machine), supports interpreters for Solaris, Microsoft, and Linux platforms. Interpreters have also been created for devices that have Windows CE or PalmOS as operating systems.

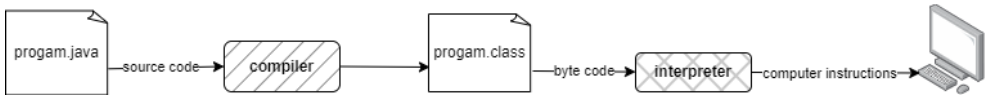



Figure 2.2: The JIT Compilation Process

In the Figure 2.1, we can see the two events of the JIT compilation process. First, the Java source code is compiled into byte code, which is platform-independent. As represented in Figure 2.2, at runtime, the byte code is converted into native machine code by the JIT compiler, which optimizes the code for the specific machine architecture. This results in faster program execution

since the native code can be executed directly by the machine, rather than being interpreted every time the program is run.

## 2.2 Advantages of Java

In this section, we will explore some advantages of Java that try to answer the natural question: why use Java when we have other OOP languages available?

Java is an object-oriented programming language that is known for its simplicity, portability, and robustness. Some of its key features and characteristics include:
- Platform independence: Java programs can run on different operating systems and hardware architectures without modification, thanks to the Java Virtual Machine (JVM).
- Object-oriented programming: Java is designed around the concept of objects, which encapsulate data and behavior in a reusable and modular way.
- Garbage collection: Java's garbage collector automatically frees up memory that is no longer being used, helping to prevent memory leaks and other memory-related issues.
- Security: Java's security model helps protect against malicious code and other security threats, making it a popular choice for web-based applications.
- Rich class libraries: Java comes with a large set of pre-built class libraries that provide support for tasks such as input/output, networking, and graphics.

1. Write once, run anywhere. This is the core concept on which the Java platform was built. In other words, once the application is written, it will run on any platform that supports Java without modification. This is an advantage over other languages that need to be rewritten (most of the time completely) to run on other operating systems.

2. Security. The platform allows users to download code over the network in a secure environment: unsafe code cannot infect the host system, cannot read/write files on the hard disk, etc. This ability made Java unique until the emergence of other competing platforms (such as .NET).

3. Network-oriented programming. Another principle of Sun is that "The network is the computer". Those who designed Java believed in the importance of communication over the network and had this in mind: Java facilitates the use of resources over the network and creates architectures on multiple levels.

4. Dynamic programs. Programs written in Java are easy to extend because the organization is modular, namely on classes. Classes are stored in separate files and loaded by the interpreter only when and if necessary. Thus, a Java application appears as an interaction between independent software components. This feature is vastly superior to applications consisting of monolithic code blocks.

5. Performance. The Java virtual machine runs a program by interpreting portable byte-code instructions. This architecture means that Java programs are slower than C, C++, which are compiled using native code. However, for efficiency, certain portions of Java, such as

string manipulation, use native code instructions. This disadvantage has been improved from version to version.

In conclusion, Java has many advantages over other OOP languages, including its platform-independence, security, network-oriented programming capabilities, modular organization, and performance. These advantages make it a preferred choice for many software developers.

## 2.3 What Can Java Offer?

Java offers a variety of tools and libraries that a programmer needs, such as tools for compiling, running, monitoring, debugging, and documenting. In general, we will use the javac compiler, the java program for running Java applications, and javadoc for documentation.

Application Programming Interface (API) is the core set of Java functions. It offers a series of useful classes that you will use in your applications. This core is very large, and to get an idea of what it contains, we have the image below. Figure 2.3 presents two concepts: JRE (Java Runtime Environment) and JDK (Java Development Kit).

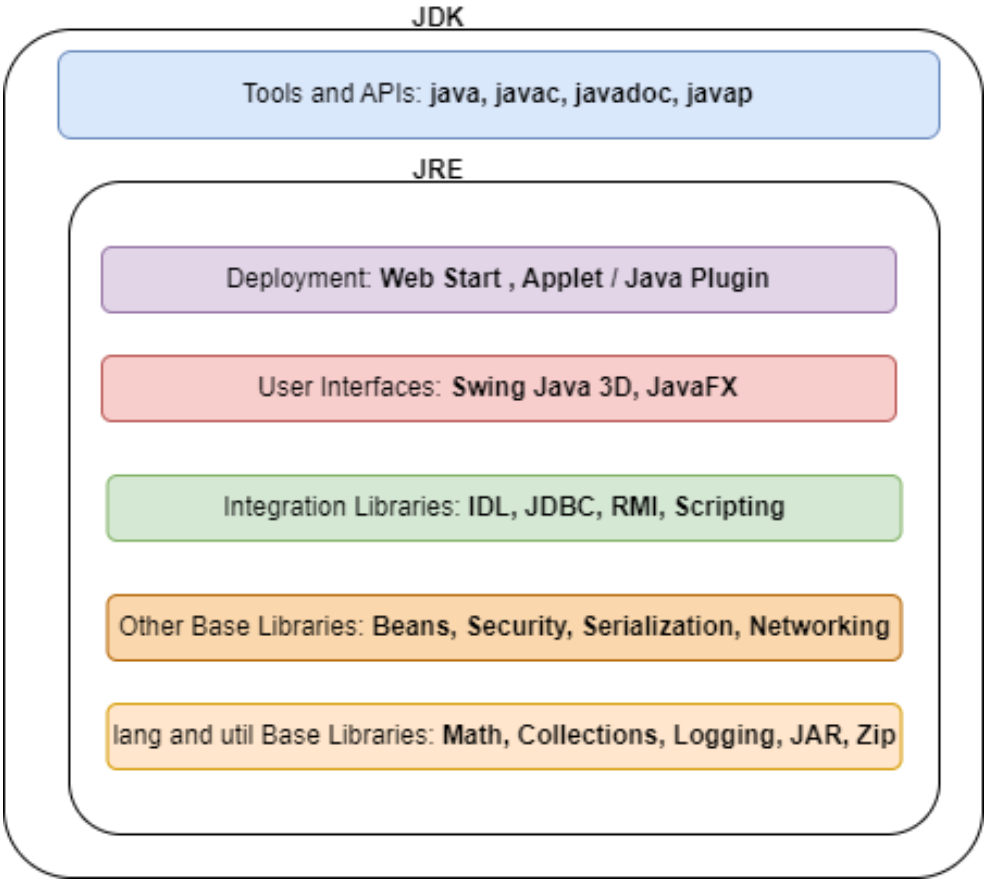



Figure 2.3: The JIT Compilation Process

JRE provides libraries, the Java virtual machine, and other components necessary for running applications and applets. This environment can be redistributed with applications to give them autonomy.

JDK includes JRE and tools such as compilers or debuggers that are useful for developers.

Tools for graphical interfaces: Swing and Java 2D are used to create GUI (Graphical User Interfaces) and offer facilities for a beautiful design (and more) of applications.

Libraries for integration: For example, Java IDL (Interface Definition Language) API, JDBC (Java Database Connectivity), Java Naming and Directory InterfaceTM ("J.N.D.I.") API, Java Remote Method Invocation, etc. These libraries allow access to databases or remote object manipulation.

## 2.4 A first example

Obtaining the JDK (Java Development Kit) Before compiling or running Java programs, we need to have this environment installed on computers. For this, we need to install the JDK from Sun (there are other packages offered by other companies, but we will use this one because it is free and is the main authority of the Java language). It can be downloaded from Oracle's web page.

```
A first Java program example
/*
This is the first program
That will be written in a file Example.java
*/
class Example {
// To start a Java program we need a main function.
    public static void main(String args[]) {
    System.out.println("Hello Java!");
    }
}
```

The steps to follow in creating a Java program will always be:

1. Writing the program.
2. Compiling it.
3. Running the program.

Obviously, things will get complicated and the steps can be detailed very much. In the following, we will briefly describe what happens and some indications for each of the three steps.

When writing Java programs, any text editor can be used, such as Notepad if working on Windows, or Joe, Nano, or Vi if working on Linux. Additionally, IDEs (Integrated Development Environments) can be used when applications become more complex.

For most programming languages, the file names containing the source code can be any name. However, in Java, the file name containing the source code is important. For example, in this case, it should be Example.java. In Java, a Java file is called a compilation unit. It is a text

file that contains one or more class definitions. The compiler requires this file to have the .java extension. The file name is Example, and this is not a coincidence with the class name. In Java, everything must be within a class. The name of the public class must be the same as the name of the file in which that class is located. Note that Java, like C, is case sensitive, meaning Example is different from example or eXample.

Compiling a Java program is done by running the javac compiler and specifying the name of the source file:

```
C:\>javac Example.java
```

The javac compiler creates a Example.class file that contains the bytecode corresponding to the source code in Example.java. Note that this bytecode is not directly executable. It will be executed by the JVM. To run the program, we need to use the Java interpreter, namely java:

```
C:\>java Example
```

When Java code is compiled, each individual class is placed in a file named after the class, using the .class extension. Therefore, it is a good idea to give the source files the same name as the class they contain. When we run the interpreter, it will look for the Example file with the .class extension. It will find it and execute the code contained in that class.

Explanation of the first program:

```
/*
This is the first program
Which will be written in a file called Example.java
This is a multi-line comment that begins with / and ends with */. Any
    characters
between these two symbols will be ignored during compilation.
```

Another way to comment code is on a single line as shown below:

```
int i =0;//This is a comment on this line only
```

Next, we have:

```
class Example {
```

This line uses the keyword ”class,” indicating the beginning of the definition of a new class. In Java, a class is the basic unit of encapsulation. The class will be defined in the block delimited by characters ’{’ and ’}’. We will not go into details about the content of a class at this point, as that is covered in the next chapter of the course.

The next line is a comment on a single line (referring to the function that will follow).

The next line is the main method or entry point:

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

It is called this because it represents the function that is first executed when the Java interpreter runs the program. All Java applications start execution with the call to main().

The keyword ”public” is an access specifier. The keywords of a language are those words that ”make up” the language and define various instructions or operators or data types, etc.

An access specifier determines how other classes in the program can access members of a class. A member of a class can be considered a variable or a function, for example.

The ”static” keyword allows the call to main() to be made without needing to instantiate the class to which this function belongs.

The ”void” keyword specifies the data type returned by the main() function, in this case, nothing, meaning it tells the compiler that the function will not return any value.

Within the parentheses of the main() function is the parameter list, in this case, ”String args[]”. This declaration ”String args[]” means the collection of objects of type String (a character string).

The last character of this line is ””, indicating that the body of the main function begins. The body of the function ends with ”” similar to the ”body” of the class.

The next part of the code is:

```
System.out.println("Hello Java!");
```

This has the effect - at run-time - of displaying the message ”Hello Java!” The display occurs due to the already defined, or predefined, function ’println()’. However, the line starts with ’System.out...’. At this point, we can say that System is a predefined class that provides access to the system, and out is the output stream connected to the console. In practice, System.out is an object that encapsulates console output. The console is not commonly used in real Java programs or applets, but it is useful in the Java learning process.

## 2.5 Syntax errors, run-time errors

When writing source code, especially if you are a beginner, it is possible to inadvertently or unknowingly omit certain portions of the code. Fortunately, if something is not correct, the compiler will report these problems as syntax errors. The compiler will try to ”make sense” of the source code, regardless of what it consists of. Therefore, the reported error does not usually reflect the cause of the problem!

For example, if we omit the ’{’ character in

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

The following message will appear when compiling:

```
Example.java: 7 ';' expected
Public static void main(String[] args)
Example.java:10: class, interface or enum expected
}
^
2 errors
```

Obviously, something is wrong with the message because it is not missing ”;” but ”{”. The idea is that when the program contains a syntax error, the message should not be interpreted word for word. We need to look at the context in which the error occurs, the lines around the location where the error is indicated, and deduce the real cause of it. Refactoring the code and the previous mistake, let’s replace the message display with this code:

```
System.out.println(args[1]);
```

What does this code achieve? Display the second argument passed to the program: the arguments are a string that will be written at runtime. We compile and run:

```
java Example
```

The following error will appear:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundException : 1 at
Example.mainExample.java:8
```

This type of exception occurs at run-time (hence the name run-time error) and is due to unforeseen situations by the programmer. In this case, it is assumed that we have arguments at run-time, when in reality, the argument string is empty. Accessing the argument with number 1 is thus an error! We will explain the full meaning of this error at the appropriate time.

## 2.6 Role of Java in Software Development

Java has become one of the most widely used programming languages in the world, thanks to its versatility and popularity. It is used to create a wide range of applications, from simple command-line tools to complex web applications and mobile apps.

Some of the key areas where Java is used include:

1. Enterprise software: Java is a popular choice for creating large-scale enterprise applications, thanks to its scalability, reliability, and robustness.

2. Web development: Java is widely used for web development, particularly for building server-side applications and web services.

3. Mobile app development: Java is used for developing Android apps, one of the most popular mobile operating systems in the world.

4. Scientific computing: Java is increasingly being used in scientific computing applications, thanks to its performance and scalability.

Overall, Java is a versatile and powerful programming language that has found widespread use in many different areas of software development.

## 2.7 Java Virtual Machine

This section describes the Java Virtual Machine (JVM), which provides a runtime environment for Java programs to run on different platforms. It explains how the JVM works, its advantages, and its key features, including memory management and security.

The Java Virtual Machine (JVM) is a software implementation that provides a runtime environment for Java programs to run on different platforms. It is the cornerstone of Java's "write once, run anywhere" promise, which means that Java code can be compiled into bytecode that

can be executed on any machine with a JVM installed, regardless of the underlying hardware and operating system.

The JVM is responsible for executing Java programs by interpreting bytecode and translating it into machine code that can be executed by the underlying hardware. It also provides a number of key features that make Java programs safe, efficient, and platform-independent.

One of the key advantages of the JVM is its memory management system, which automatically manages memory allocation and deallocation for Java programs. The JVM uses a garbage collector to automatically reclaim memory that is no longer being used by the program, which helps prevent memory leaks and other memory-related issues. This makes Java programs more reliable and less prone to crashes and other errors caused by memory issues.

Another important feature of the JVM is its security model, which provides a sandboxed environment for Java programs to run in. This means that Java programs are isolated from the underlying operating system and cannot directly access system resources without permission. This helps prevent malicious code from damaging the underlying system and makes Java programs more secure and trustworthy.

The JVM also provides other key features that make Java programs efficient and scalable, such as just-in-time (JIT) compilation, which compiles bytecode into machine code on the fly for faster execution, and support for multithreading, which allows Java programs to run multiple threads concurrently for better performance.

Overall, the JVM is a powerful and essential component of the Java platform that provides a secure, efficient, and platform-independent runtime environment for Java programs.

## 2.8 Java Development Environment

In this section, we will discuss the tools and software that are commonly used for developing Java applications. We will cover the Java Development Kit (JDK), Integrated Development Environments (IDEs), and other tools that can help you write, test, and debug Java code.

### 2.8.1 Java Development Kit (JDK)

The Java Development Kit (JDK) is a set of tools and software that are used for developing Java applications. The JDK includes the Java Runtime Environment (JRE), the Java compiler, and other tools such as the Java debugger and the Java documentation generator.

The JRE is the environment in which Java applications run. It includes the Java Virtual Machine (JVM) and the core class libraries. The Java compiler is used to compile Java source code into bytecode, which can then be run on any platform that supports the JVM.

### 2.8.2 Integrated Development Environments (IDEs)

Integrated Development Environments (IDEs) are software applications that provide a comprehensive set of tools for developing software. IDEs are particularly useful for developing large software projects, as they can help manage the complexity of the codebase.

Some popular Java IDEs include:
- Eclipse: Eclipse is an open-source IDE that is widely used for Java development. It provides a wide range of features, including code highlighting, debugging tools, and refactoring support.
- NetBeans: NetBeans is another popular open-source IDE for Java development. It provides a user-friendly interface and a wide range of features for developing Java applications.
- IntelliJ IDEA: IntelliJ IDEA is a commercial IDE that is known for its advanced features and excellent support for Java development.

### 2.8.3 Other Java Development Tools

In addition to the JDK and IDEs, there are many other tools and software applications that can be used for Java development. Some of these include:
- Build tools: Build tools such as Apache Maven and Gradle can help automate the process of building and packaging Java applications.
- Testing frameworks: Testing frameworks such as JUnit and TestNG can be used to write and run automated tests for Java applications.
- Profiling tools: Profiling tools such as Java VisualVM can be used to analyze the performance of Java applications and identify potential performance bottlenecks.
- Version control systems: Version control systems such as Git and Subversion can be used to manage the source code for Java applications.

Overall, the Java development environment includes a wide range of tools and software applications that can help you write, test, and debug Java code. By becoming familiar with these tools, you can improve your productivity and efficiency as a Java developer.

## Exercises

### Exercise 1: Basic Calculator

Create a Java program that performs basic arithmetic operations such as addition, subtraction, multiplication, and division on two user-input numbers. Prompt the user for input, perform the calculations, and display the results.

### Exercise 2: String Manipulation

Write a Java program that takes a user-input string and performs the following tasks:

- Print the length of the string.
- Convert the string to uppercase.
- Check if the string contains the word ”Java” and print the result.

## Exercise 3: Temperature Converter

Develop a Java program that converts temperatures between Fahrenheit and Celsius. Ask the user for input in one unit and convert it to the other unit using the conversion formulas.

## Exercise 4: Odd or Even Numbers

Create a Java program that prompts the user to enter an integer. Determine if the entered number is odd or even and print the result.

## Exercise 5: Password Validator

Design a Java program that checks the validity of a password entered by the user. Implement the following rules:

- The password must be at least 8 characters long.
- It must contain at least one uppercase letter.
- It must contain at least one lowercase letter.
- It must contain at least one digit (0-9).

# Chapter 3

# Basic Java Programming: Syntax and Structure

In this chapter, we will cover the basics of Java programming. We will start with the syntax and structure of Java code, and then move on to data types, variables, operators, and control structures J. (2018).

Before we start exploring Java syntax, let's first study the overall structure of a Java program. Java consists of one or more compilation units, which are files containing classes. Each unit can start with an optional package keyword, followed by import declarations. These, similar to the include directive in C, allow the use of classes from other packages. We will discuss these aspects later.

Then comes the definition of one or more classes, interfaces, and more recently, enum structures. Within classes, we can define fields (variables), methods, or constructors. All of these are called class members. Most methods will contain instructions that include expressions, operators, data types, etc. The approach from here on will be exactly from the basic units towards packages in the end. Packages contain classes, and classes contain methods. This inclusion relationship is marked by a dot. The difference is that in the case of packages, it signifies a relationship of inclusion between directories/folders, while in the case of classes/methods, it indicates their inclusion in the same file.

Java code is structured into classes, which contain methods. Each method is a block of code that performs a specific task. The syntax of a Java method is as follows:

```
1 public static void methodName() {
2   // code goes here
3 }
```

In this example, the method is named `methodName`, and it has the `public`, `static`, and `void` modifiers. These modifiers determine the access level, behavior, and return type of the method.

## 3.1 Data Types and Variables

Java has several built-in data types, including `int`, `double`, `boolean`, and `String`. Variables are used to store values of these data types. The syntax for declaring a variable is as follows:

```
dataType variableName = value;
```

For example, to declare an integer variable named `num` with the value 42, we would write:

```
int num = 42;
```

Variable is an element of a certain data type identified by a name, used for storing data. The variable name, also called identifier, is composed of a series of characters that starts with a letter. The data type of the variable determines what values that variable can take.

The declaration will be in the form: type name. In addition to name and type, a variable also has a scope. The section of code where that variable can be used is called the variable's scope.

Let's analyze the code below for a better understanding of what was described above.

```
public class Variabile {
    public static void main(String args[]) {
        // integer
        byte largestByte;
        largestByte = Byte.MAX_VALUE;
        int largestInteger = Integer.MAX_VALUE;
        // real
        float largestFloat = Float.MAX_VALUE;
        double largestDouble = Double.MAX_VALUE;
        // afisarea
        System.out.println("Valoarea maxima byte este " + largestByte);
        System.out.println("Valoarea maxima integer value este " +
    largestInteger);
        System.out.println("Valoarea maxima float value este " +
    largestFloat);
        System.out.println("Valoarea maxima double value este " +
    largestDouble);
    }
}
```

The Variable class contains a main() method and no variables. Yes, that's correct, no variables, because the scope where the four variables are visible is the main() function.

Let's take the declaration of the byte variable largestByte; By this instruction, a variable named largestByte is declared, which has the data type byte. This data type is a signed integer value between -128 and 127. We will detail the primitive data types in Table 3.1. Java contains two categories of data types: reference and primitive. A variable of a primitive type can only contain a single value according to that type and having the format of that type. Obviously, reference types are more complex and we can say that they encompass primitive types.

Classes, arrays, and interfaces are reference data types. The value of a reference variable is the address of an object. A reference is also known as an object, or a memory address in other languages, but in Java, we do not have the ability to directly access the memory area as in C.

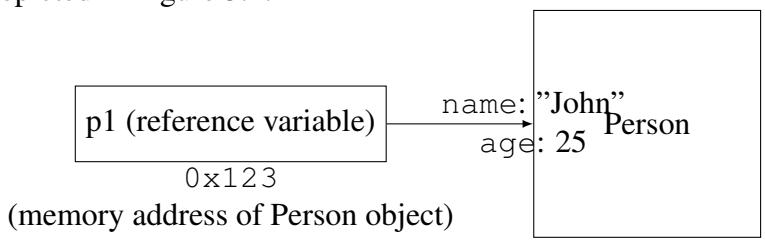
A reference to an object in Java is a variable that stores the memory address of an object created on the heap. Here's an example diagram of a reference variable in Java:

| Type | Size (bits) | Range |
|---|---|---|
| `byte` | 8 | $-2^7$ to $2^7 - 1$ |
| `short` | 16 | $-2^{15}$ to $2^{15} - 1$ |
| `int` | 32 | $-2^{31}$ to $2^{31} - 1$ |
| `long` | 64 | $-2^{63}$ to $2^{63} - 1$ |
| `float` | 32 | $\pm 3.40282347 \times 10^{38}$ |
| `double` | 64 | $\pm 1.797693134 \times 10^{308}$ |
| `char` | 16 | Unicode characters |
| `boolean` | 8 | `true` or `false` |

Table 3.1: Primitive Data Types in Java

```
Person p1 = new Person("John", 25);
```

In this example, *p1* is a reference variable that is storing the memory address of a new Person object created on the heap. The Person object itself contains two fields: name and age. The reference variable *p1* points to this object on the heap, allowing us to access and modify its fields as depicted in Figure 3.1.




**Variable Naming Rules**

To be considered a valid variable name, it must meet the following criteria:

1. Must start with a letter and be composed of Unicode characters
2. Cannot be a reserved keyword
3. Must be unique within its scope of visibility

**Variable scope**

The scope of a variable is the region in the program where that variable can be referenced. Another purpose of scope is to determine when the system creates or destroys the memory allocated for a variable.

**Variable initialization**

Local and member variables must be initialized before they can be used. Initialization means assigning values to them:

```
int largestInteger = Integer.MAX_VALUE;
```

or

```
int largestInteger = 23;
```

The `final` keyword for variables means that a variable can no longer be modified once it has been initialized, effectively making it a constant.

```
final int largestInteger = 0;
```

## 3.2 Operators

Java supports a variety of operators, including arithmetic operators (+, -, *, /), relational operators (==, !=, <, >, <=, >=), and logical operators (&&, ||, !). These operators are used to perform calculations, compare values, and evaluate conditions.

An operator performs a function on one, two, or three operands. An operator that requires only one operand is called a unary operator. For example, ++ is a unary operator that increments the value of the operand by 1. An operator that requires two operands is called a binary operator. For example, = is an operator that assigns a value to the operand on the left, with the value being the operand on the right. An operator with three operands would be ?: and we will talk about it shortly.

Unary operators can have prefix, infix, or postfix notation:

- operator op - prefix notation
- op operator - postfix notation
- op1 operator op2 - infix notation

**Arithmetic Operators**
These operators help constructing the mathematic expressions represented in Table 3.2.

| Operator | Usage | Description |
|---|---|---|
| + | op1 + op2 | Adds op1 and op2; also concatenates strings |
| - | op1 - op2 | Subtracts op2 from op1 |
| * | op1 * op2 | Multiplies op1 with op2 |
| / | op1 / op2 | Divides op1 by op2 |
| % | op1 % op2 | Calculates the remainder of op1 divided by op2 |

Table 3.2: Arithmetic operators

**Relational Operators**
These compare two values and determine the relationship between them.
**Conditional Operators**
These operators return a true or false value by evaluating the operands.
**Bitwise Operators**
These operators perform bitwise operations on the operands.
**Shift Operators**
Operator >>> fills the leftmost bits with zero, while >> fills the leftmost bits with the sign bit. These operators perform bit shifting operations on the operands.

| Operator | Usage | Description |
|---|---|---|
| $>$ | $op1 > op2$ | Returns true if $op1$ is greater than $op2$ |
| $>=$ | $op1 >= op2$ | Returns true if $op1$ is greater than or equal to $op2$ |
| $<$ | $op1 < op2$ | Returns true if $op1$ is less than $op2$ |
| $<=$ | $op1 <= op2$ | Returns true if $op1$ is less than or equal to $op2$ |
| $==$ | $op1 == op2$ | Returns true if $op1$ is equal to $op2$ |
| $!=$ | $op1! = op2$ | Returns true if $op1$ is not equal to $op2$ |

Table 3.3: Relational Operators

| Operator | Usage | Description |
|---|---|---|
| && | $op1\&\&op2$ | Returns true if both $op1$ and $op2$ are true; evaluates conditional $op2$ |
| \|\| | $op1\|\|op2$ | Returns true if either $op1$ or $op2$ is true; evaluates conditional $op2$ |
| ! | $!op$ | Returns true if $op$ is false |

Table 3.4: Conditional Operators

## 3.3 Control Structures

Control structures are used to control the flow of execution in a Java program. Java supports several types of control structures, including if statements, for loops, while loops, and switch statements.

### While

The general form of the `while` statement is:

```
while (expression) {
// statement(s)
}
```

Here, `expression` is evaluated and if the resulting value is `true`, the `statement(s)` within the loop are executed. This process repeats until `expression` evaluates to `false`.

For example,

```
1 int i = 0;
2 while (i < 10) {
3     System.out.println(i++);
4 }
5 System.out.println("After the loop");
```

| Operator | Usage | Description |
|---|---|---|
| & | $op1 \& op2$ | Returns true if both $op1$ and $op2$ are boolean and both are true; always evaluates $op1$ and $op2$ |
| \| | $op1 \| op2$ | Returns true if both $op1$ and $op2$ are boolean and at least one is true; always evaluates $op1$ and $op2$ |
| ^ | $op1 \hat{} op2$ | Returns true if $op1$ and $op2$ are different, i.e., an XOR operation on the bits |

Table 3.5: Bitwise Operators

| Operator | Usage | Description |
|---|---|---|
| $<<$ | $op1 << op2$ | Shift the bits of $op1$ to the left by the no of bits specified by $op2$ |
| $>>$ | $op1 >> op2$ | Shift the bits of $op1$ to the right by the no of bits specified by $op2$ |
| $>>>$ | $op1 >>> op2$ | Shift the bits of $op1$ to the right by the no of bits specified by $op2$ |

Table 3.6: Shift operators

will print the values 0 through 9, and then print the message "After the loop".

The `do-while` statement is similar to the `while` statement, except that the `expression` is evaluated at the end of the loop. This means that the statements within the loop are executed at least once.

The `for` statement allows for compact iteration through a range of values, and is typically used when the number of iterations is known in advance. The general form of the `for` statement is:

```
for (initialization; termination condition; increment) {
    // statement(s)
}
```

Here, the `initialization` code is executed only once, at the beginning of the loop. The `termination condition` is evaluated at the beginning of each loop iteration, and the loop terminates if it evaluates to `false`. The `increment` statement(s) are executed at the end of each loop iteration.

For example,

```
1 for (int i = 0, j = 1; (i < 10 || j < 16); i++, j += 2) {
2     System.out.println(i + " " + j);
3 }
4 System.out.println("After the loop");
```

will print the values of `i` from 0 to 8 and the odd values of `j` from 1 to 15, and then print the message "After the loop".

## If-Else

The `if` statement allows for conditional execution of statements. The basic form of the `if` statement is:

```
if (expression) {
    // statement(s)
}
```

Here, if `expression` evaluates to `true`, the `statement(s)` within the block are executed. If `expression` evaluates to `false`, the `statement(s)` are skipped.

For example,

```
if (response == OK) {
    // code to perform OK action
} else {
    // code to perform Cancel action
}
```

If `response` is equal to `OK`, the code to perform the OK action is executed. Otherwise, the code to perform the Cancel action is executed.

## Other control statements

### Break

Sometimes, it is necessary to force the exit from a loop (for, while) using the **break** statement. In the following examples, as seen in the previous examples, we have created a **for** loop, but this time there is no continuation condition, so the exit can only occur within this **for** loop.

```
class BreakSample {
    public static void main(String args[]) {
        int num;
        num = 10;
        // continues "forever",
        //the continuation condition does not exist
        for(int i = 0; ; i++) {
            if(i >= num) break; // i reaches 10, we exit the loop
            System.out.print(i + " ");
        }
        System.out.println("The loop is over.");
    }
}
```

This program will generate the following sequence:

```
0 1 2 3 4 5 6 7 8 9 The loop is over.
```

As you can see, this type of instruction occurs in conjunction with conditional statements, otherwise the loop would only be executed once, eliminating its repetitive character. Other purposes of using this instruction can be, for example, the repetitive reading of characters: when

what we read is equal in value to the character that marks the end of the reading (for example 'q'), we immediately exit the repetitive loop in which the reading takes place.

The **break** instruction can also be used as a jump instruction to a specific label. Java does not have a classic **goto** instruction because it allows unstructured logic flow. Programs that use this type of instruction are difficult to follow and maintain, which is why the exposure is purely informative, and I do not recommend using **break** for this purpose. The general form of a **break** with a **goto** sense is:

```
break label;
```

In this case, the label is the name of the label that identifies a block of instructions. When the **break** instruction with the **label** is executed, control is transferred to that block indicated by the label name.

### Continue

This instruction is used in a repetitive loop, allowing, if we do not want to continue with the instructions in that repetitive loop, the instructions that follow **continue** to skip to the next step. Practically, it "jumps" to the next step in the repetitive instruction, ignoring what happens after **continue**.

For example, if we want to display even numbers between 0 and 50:

```
class ContinueSample {
    public static void main(String args[]) {
        int i;
        // display even numbers between 0 and 50
        for(i = 0; i <= 50; i++) {
            if((i % 2) != 0) continue; // if it is not even,
                                       //go to the next step in for
            System.out.println(i);
        }
    }
}
```

In this example, only even numbers will be displayed, because: the conditional statement is entered only if the remainder of the current number divided by 2 is zero. When entered (so when the number is odd), **continue** is executed, which means to go to the next.

### Nested Repetitive Loops

Before concluding this subchapter on basic instructions, let's exemplify the use of nested repetitive loops. In the following example, we calculate the factors of each number from 2 to 50 as follows: we iterate through each number from 2 to 50; at each step, we are in a position to calculate the factors of the current number, let's say X. At this point, another loop iterates through the numbers up to X (this can be optimized, of course). In the innermost loop, we check the remainder of dividing X by the numbers in the interval [2..X). If it's zero, it means we found a factor of X:

```
class Factors {
    public static void main(String args[]) {
        for(int i = 2; i <= 50; i++) {
            System.out.print("Factors of " + i + ": ");
            for(int j = 2; j < i; j++)
                if((i % j) == 0) System.out.print(j + " ");
            System.out.println();
        }
    }
}
```

**Conditional Operator**

The conditional operator ?: is a ternary operator inherited from C. It allows embedding a condition within an expression. The first operand is separated from the second operand by ?, while the second operand is separated from the third operand by :. The first operand must be of boolean type. The second and third operands can be of any data type, as long as they are of the same data type or convertible to the same data type.

How is this operator evaluated? The first operand is evaluated; if it's true, the operator evaluates the second operand and uses its value. If it's false, the operator evaluates the third operand and returns its value. For example:

```
int max = (x > y) ? x : y;
String name = (name != null) ? name : "unknown";
```

## 3.4 Arrays

Arrays in Java are objects that store multiple variables of the same type. They are ordered and can be accessed by an index.

### 3.4.1 Array Declaration and Initialization

Arrays are declared by specifying the type of their elements followed by square brackets.

```
int[] myArray;
```

Arrays can be initialized at the time of declaration.

```
int[] myArray = {1, 2, 3, 4, 5};
```

Or after the declaration.

```
int[] myArray = new int[5];
myArray[0] = 1;
myArray[1] = 2;
myArray[2] = 3;
myArray[3] = 4;
myArray[4] = 5;
```

### 3.4.2 Array Length

In Java, arrays are not immutable by default; their elements can be changed once the array is initialized. However, the size of the array is **fixed** upon creation and cannot be changed. This is different from some other types of collections in Java, like $ArrayList$, which can dynamically resize. We will study these collections in next chapters. The length of an array can be accessed using the `length` property.

```
int arrayLength = myArray.length;
```

## 3.5 Understanding Strings in Java

Strings in Java are more than just a simple data type; they are objects that represent a sequence of characters (‘char‘). One of the distinct features of Java strings is their **immutability**, which means once a string is created, its content cannot be changed.

### 3.5.1 Creating Strings

Strings can be created using string literals or with the `new` keyword. Using string literals is more common and efficient:

```
// Using string literal (recommended)
String str1 = "Hello";

// Using new keyword
String str2 = new String("Hello");
```

### 3.5.2 Common String Methods

Java’s `String` class provides a variety of methods for various string manipulations. However, it is crucial to remember that these methods return a new string and do not modify the original. Some of these methods include:

- `length()`: Returns the length of the string.
- `charAt(int index)`: Returns the character at the specified index.
- `substring(int beginIndex, int endIndex)`: Returns a new string that is a substring of the original string.
- `toUpperCase()`: Converts the string to upper case and returns it.
- `toLowerCase()`: Converts the string to lower case and returns it.

```
String original = "Hello, World!";
int length = original.length();
char firstChar = original.charAt(0);
String hello = original.substring(0, 5);

System.out.println("Length: " + length);
System.out.println("First character: " + firstChar);
System.out.println("Substring: " + hello);
```

This code would output:

```
Length: 13
First character: H
Substring: Hello
```

### 3.5.3 Immutability of Strings

The `String` class in Java is designed with immutability in mind. This implies that once a `String` object is created, it is not possible to alter its contents.

```
String immutable = "Original";
String modified = immutable.replace('i', 'o');

System.out.println("Immutable: " + immutable);  // Output: Original
System.out.println("Modified: " + modified);  // Output: Oroginal
```

Even after the `replace()` operation, the original string remains unchanged, thereby adhering to the concept of string immutability in Java.

The immutability of strings has several advantages, such as:

1. **Security**: The data within the string is secure as it cannot be changed.
2. **Synchronization and Concurrency**: Immutable objects are naturally thread-safe.
3. **Caching**: Since they always have the same state, immutable objects are good candidates for caching.

However, if you intend to modify strings frequently, consider using `StringBuilder` or `StringBuffer` classes for more efficient string manipulation.

### 3.5.4 Common String Methods

The ‘String‘ class provides a plethora of utility methods to perform operations on strings. Here are some commonly used methods, with examples to demonstrate how they abide by the principle of immutability.

- **concat(String str)**: This method appends the specified string at the end of the current string.

```
String greeting = "Hello";
String completeGreeting = greeting.concat(", World!");
System.out.println("Original: " + greeting);  // Output: Hello
System.out.println("Modified: " + completeGreeting);  // Output:
  Hello, World!

```

Notice how the original string remains unchanged, reinforcing its immutable nature.

- **trim()**: This method returns a new string with the leading and trailing white spaces removed.

```
String spacey = "  Trim me  ";
String trimmed = spacey.trim();
System.out.println("Original: '" + spacey + "'");  // Output: "  Trim
  me  "
System.out.println("Trimmed: '" + trimmed + "'");  // Output: "Trim
  me"

```

- **indexOf(int ch) or indexOf(String str)**: This method returns the index of the first occurrence of the character or substring. If it doesn't exist, the method returns -1.

```
String sentence = "Look for the letter o.";
int index = sentence.indexOf('o');
System.out.println("Index: " + index);  // Output: Index: 1

```

- **replace(char oldChar, char newChar)**: Replaces all occurrences of the specified 'oldChar' with the 'newChar'.

```
String text = "abbabba";
String replacedText = text.replace('a', 'z');
System.out.println("Original: " + text);  // Output: abbabba
System.out.println("Replaced: " + replacedText);  // Output: zbbzbbz

```

Note again how the original string is not modified, which confirms its immutable nature.

## 3.6 Switch Statements

The `switch` statement allows for a variable to be tested for equality against a list of values.

### 3.6.1 Basic Switch Statement

The switch variable can be char, byte, short, integer, or enumerable. This value will be evaluated against the *case* ones and if it matches one, then the code corresponding to that *case*, will be executed. The *break* prevents the execution of the instructions that are written below that *case* which was initally matched.

```
switch (variable) {
    case value1:
        // Code to be executed;
        break;
    case value2:
        // Code to be executed;
        break;
    default:
        // Code to be executed if variable is different from all values;
}
```

### 3.6.2 Switch with Strings

Starting from Java 7, you can use a `String` object in the `switch` statement.

```
String day = "Monday";
switch (day) {
    case "Sunday":
        System.out.println("It is weekend!");
        break;
    case "Saturday":
        System.out.println("It is weekend!");
        break;
    default:
        System.out.println("It is a weekday.");
}
```

## 3.7 Methods

Methods in Java serve as the building blocks of object-oriented programming. They encapsulate specific functionalities into reusable pieces of code.

### 3.7.1 Method Declaration

The basic syntax for declaring a method in Java is as follows:

```
[access_modifier] [static] [return_type] methodName(parameters) {
    // code block
}
```

- **access_modifier** - Defines the visibility of the method (public, private, protected, or default).
- **static** - Indicates that the method belongs to the class rather than any specific instance of the class.
- **return_type** - Specifies what type of value the method will return.

- **methodName** - The name of the method, usually in camelCase.
- **parameters** - A list of parameters that the method accepts, separated by commas.

### 3.7.2 Method Call

To invoke a method, you use its name followed by arguments enclosed in parentheses.

```
methodName(arguments);
```

For methods with a return value, the method call is often part of an expression.

```
int result = sum(3, 4);  // Calls the sum method and stores the result in
    variable 'result'
```

### 3.7.3 Method Overloading

Method overloading in Java allows you to have multiple methods with the same name but different parameter lists. The return type can be the same or different. The compiler distinguishes these methods based on their parameter lists.

```
public static int sum(int a, int b) {
    return a + b;
}

public static double sum(double a, double b) {
    return a + b;
}
```

In the example above, the method `sum` is overloaded with two different parameter types: `int` and `double`. More on this will be reiterated in the next chapter.

### 3.7.4 Method Return

Methods can return a value, specified by the return type in the method declaration. The `return` keyword is used to specify the value that a method returns.

```
public static int square(int a) {
    return a * a;
}
```

The `square` method takes an integer as an argument and returns the square of that integer.

### 3.7.5 Recursive Methods

In Java, methods can be recursive, meaning a method can call itself either directly or indirectly. Recursive methods are often used for problems that have a repetitive structure, such as calculating factorials or traversing trees.

```java
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
}
```

Here, the `factorial` method calls itself to find the factorial of a number. The base case (`n == 0`) ensures that the recursion terminates.

In conclusion, understanding the basic syntax of Java is essential for any aspiring programmer. This foundation serves as a launching pad for more complex software development. By mastering key concepts such as data types, variables, operators, control structures, and loops, you gain the ability to create functional and organized programs.

Java's strict syntax rules promote consistency and reliability in coding. Variable declaration, assignment, and manipulation ensure data integrity and efficient memory usage.

# String Exercises

## Exercise 1: Common Elements in Arrays

Given two arrays of integers, find the number of common elements.

- Example: $\{12, 54, 2, 78, 36, 22, 92\}$ and $\{33, 6, 19, 86, 54, 44, 20, 78\} \Rightarrow |2|$

## Exercise 2: Palindromic Array Check

Check if the elements within an array form a palindrome.

- Example: $\{23, 56, 79, 61, 34, 61, 79, 56, 23\}$ – yes, $\{21, 44, 37, 90, 21, 44, 37\}$ – no

## Exercise 3: Extracting a Row from a Matrix

Given a matrix $M$ (with $m \times n$ rows and columns), extract a row of $m$ elements, where each element is the sum of the $n$ integers in each column of $M$.

- Example:

$$\begin{bmatrix} 1 & 1 & 2 \\ 5 & 0 & 0 \\ 2 & 3 & 3 \end{bmatrix} \Rightarrow \{8, 4, 5\}$$

## Exercise 4: Sorting an Array of Strings

Given the array of strings:

static String arr[] = {"Now", "is", "the", "time", "for", "all", "good", "men", "to", "come", "to", "the", "aid", "of", "their", "country"};

1. Create another array of strings with the elements of $arr$ sorted using the `compareTo()` method of the `String` class.

2. Concatenate the strings in the $arr$ array using the "+" operator. Then concatenate using the `join()` method. The result should look like:

   String str = "Now,is,the,time,for,all,good,men,to,come,to,the,aid,of,their,country";

3. Find another method to concatenate the strings to obtain an array similar to $arr$ starting from the $str$ variable. Hint: use the `split()` method.

## Exercise 5: Converting Data Types to Strings and Back

Given the following variables:

$$\begin{aligned} doubled &= 102939939.939E + 20; \\ booleanb &= \text{true}; \\ longl &= 1232874; \\ char[]arr &= \{'a','b','c','d','e','f','g'\}; \end{aligned}$$

1. Convert them to `String` variables using the `valueOf()` method and display them in the console.

2. Try to convert the `String` variables back to their original data types.

# Chapter 4

# Object-Oriented Programming in Java

Object-oriented programming is a programming paradigm that emphasizes the use of objects to represent real-world entities and their interactions. Java is an object-oriented language, which means that all code in Java is written using objects and classes Bloch (2017).

## 4.1 Classes and Objects

In Java, a class is a blueprint for an object. It defines the properties and methods that an object of that class will have. To create an object of a class, you use the `new` keyword, followed by the name of the class and any arguments required by its constructor. Here is an example:

```
public class Car {
  private String make;
  private String model;
  private int year;

  public Car(String make, String model, int year) {
    this.make = make;
    this.model = model;
    this.year = year;
  }

  public void start() {
    // code to start the car
  }

  // additional methods go here
}

Car myCar = new Car("Toyota", "Corolla", 2020);
```

In this example, we have defined a class called `Car` with three properties (`make`, `model`, and `year`) and two methods (`start` and an unspecified number of additional methods). We have also created an object of this class called `myCar`.

**Objects** are the main concept in object-oriented technology. Real-world objects that surround us, such as tables, televisions, tools, animals, etc., have two characteristics in common: state and behavior. For example, animals have states (name, color, species) and behavior (mode of movement, breathing, feeding). Bicycles have states (current speed, weight, pedaling cadence) and behavior (acceleration, braking, changing gears).

Objects are modeled in programming by one or more variables, and the behavior of objects is modeled by methods as presented in Figure 4.1.

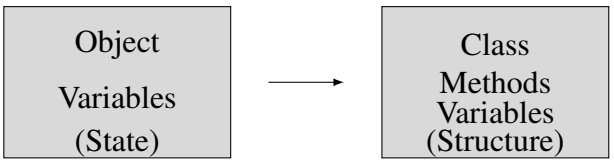



Figure 4.1: OOP Concepts

Everything known about an object represents its state expressed through variables, and what an object can do represents its behavior represented through methods. For example, a software-modeled object that models a bicycle will have variables that indicate the current state of the bicycle: speed of 10 km/h, pedaling cadence of 90 rpm, and the current gear (third gear). In addition to variables, a bicycle represented in software can have methods for braking, changing gears, etc. However, it cannot have a method for changing the speed because the speed is a consequence of pedaling cadence, braking (or not), and the slope. It could have a method for playing back the speed calculated based on these factors. These methods are called instance methods because they evaluate or modify the state of a particular bicycle, not all bicycles.

The diagrams above indicate that variables form the core, the nucleus of objects, and methods surround this nucleus. Packaging an object's variables to protect the information they contain is called encapsulation.

**Class**

In the real world, many objects can be categorized as being of the same type. For example, the reader's car is roughly the same as any other car in the world. They have wheels on which they move, the engine causes the movement, and some form of fuel (in one form or another) is needed to power the engine. Similarly, a bicycle will have wheels, a pedaling system, one or more gears, etc. We can say that a particular bicycle – for example, the Atomik Mountain Bike – is an instance of the Bicycle class. We have built an object with certain characteristics (color, resistance) but which looks and behaves like a bicycle.

## 4.2 Functions

In the examples above, the `Automobile` class contains data in the form of variables, but no methods. Although classes that only contain variables are perfectly valid, most will contain methods to ensure a logical flow.

Methods are functions, subroutines that manipulate and process data defined in classes to ensure desired logic. A method consists of its signature and body. The signature of a method refers to its return type, name, and parameter list, while the body refers to the block of statements executed whenever the function is called.

A method contains one or more instructions that, together, perform a particular task. Methods have names by which they are identified within the class, and parentheses always follow the name. Within the parentheses, method parameters (variables of various types) are declared. These variables take on values at the time the method is called. A general method has the form:

```
return_type name(parameter_list)
{
    // Method body
}
```

The *return_type* represents any valid data type, including classes you create; examples of data types include *int*, *Integer*, *String*, or *Automobile*. If a method does not return any data type, it will be *void*. The method name can be any valid identifier and follows the same naming conventions as variables (see variable naming rules). The parameter list is a sequence of variable/parameter declarations that will be visible within the method, separated by commas:

```
int myFunction(double param1, Integer param2, Automobile minivan)
{
    // ... statements in the method body
    return param2;
}
```

Let's take the `Car` class from earlier, to which we add a method called `PrintProperties` in addition to the already existing method. This is declared static since it will be called from a static method (`main`). Another way to "solve" this problem would have been to create a new `CarDemo` object in main and call the method through that object as we will see shortly after the example below:

```
static void PrintProperties(Automobile vehicle) {
    // Check for null objects, otherwise there will be an error accessing
    its members
    if (vehicle == null) {
        System.out.println("Please enter an instantiated object!");
        return;
    }
    System.out.println("The " + vehicle.model + " car has a maximum speed of
    " + vehicle.speed);
}
```

## 4.3 Encapsulation

Encapsulation is the process of hiding the internal details of an object and exposing only the necessary information to the outside world. In Java, encapsulation is achieved through the use

of access modifiers (`public`, `private`, `protected`, and `default`) and getter and setter methods.

This section explains the concepts of classes and objects in Java, and how they are used to organize and encapsulate code.

### 4.3.1 Accessing members

Getters and setters are methods used to access and modify the private members of a class, respectively. They are an important aspect of encapsulation, which is the concept of hiding the implementation details of a class and exposing only the necessary functionalities to the outside world.

Getters are methods that provide access to the private data members of a class, allowing external code to read their values. They usually have no arguments and return the value of the private member.

Setters, on the other hand, are methods that modify the values of the private data members. They usually take an argument that is used to set the value of the corresponding private member. Setters can also be used to enforce constraints on the input values, such as checking for validity or ensuring that certain values are within a specific range.

```
public class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Using getters and setters provides several benefits, including improved security, better maintainability, and increased flexibility. They allow us to control access to the private members of a class, preventing unwanted changes and ensuring data integrity. Additionally, they make it easier to modify the implementation details of a class without affecting external code that uses it.

In this example, we have a Person class with two private instance variables: name and age. We've defined public getter and setter methods for each of these variables, using the naming convention getVariableName() and setVariableName(value).

The getters return the current value of the variable, while the setters set the value of the variable to the given parameter. The use of the this keyword inside the setter methods ensures that we're updating the instance variable of the current object.

With these getters and setters in place, we can now safely access and modify the private instance variables of a Person object from outside the class:

```
Person person = new Person();
person.setName("Alice");
person.setAge(25);

String name = person.getName(); // returns "Alice"
int age = person.getAge(); // returns 25
```

### 4.3.2 Constructors

In the previous examples, in those where we used the Automobile class, we initialized the class members such as model and speed in the function in which we also created the new minivan object:

```
minivan.model = "Logan MCV";
minivan.viteza = 180;
```

This way of working is not professional, as it requires extra attention from the programmer, who can omit initializing some members (what if we have ten members in the same class?). A constructor is a method that initializes an object when it is created (with the new operator). It has the same name as the class and does not return anything. Normally, constructors are used to create initialization procedures to "shape" the new object. Below is an example of a constructor for the Automobile class:

```
class Automobile
{
    String model; //combi, SUV, MCV, truck etc
    int consum; // 5 .. 15
    int speed; //maximum speed that can be reached: 180 .. 340

    //This is the constructor of the Automobile class called when
    instantiating
    //a new object of type Automobile
    public Automobile()
    {
        model = "Unknown";
        consum = 0;
        speed = 0;
    }
}
...
public class AutomobileDemo
{
    public static void main(String[] args)
    {
```

```
        Automobile minivan = new Automobile(); //This is where we call the constructor
    }
}
```

In the previous example, a constructor without parameters was used. Although it is OK, in many situations, there are certain times when we need constructors through which to manipulate the initial values of an object. For this, we will pass these values as parameters of a constructor of the same class, just as it happens when calling functions with various parameters:

```
class Automobile
{
    String model; //combi, SUV, MCV, truck etc
    int consum; // 5 .. 15
    int speed; //maximum speed that can be reached: 180 .. 340

    //constructor without parameters
    public Automobile()
    {
        model = "Unknown";
        consum = 0;
        speed = 0;
    }

    //constructor with one parameter
    public Automobile(String initialModel)
    {
        model = initialModel;
        consum = 0;
        speed = 0;
    }

    //constructor with two parameters
    public Automobile(String initialModel, int initialSpeed)
    {
        model = initialModel;
        consum = 0;
        speed = initialSpeed;
    }
}
public class AutomobileDemo
{
    public static void main(String[] args)
    {
        //Constructor with one parameter
        //later I also need to initialize speed separately
        Automobile minivan = new Automobile("Logan MCV");
        minivan.speed = 180;

        //Constructor with two parameters, I initialize everything
        //directly through this call
        Automobile sportsCar = new Automobile("BMW",320);
```

```
43          System.out.println(minivan.model +" has a maximum speed of " +
44              minivan.speed);
45          System.out.println(sportsCar.model +" has a maximum speed of " +
46              sportsCar.speed);
47      }
48  }
```

In the example above, we have a constructor without parameters (implicit or default) and two constructors with one and two parameters, respectively. Obviously, the data type of the parameters can vary, just like in the case of methods.

We mentioned that a constructor resembles a function. When is a constructor called? When we use the new operator to create a new object, first, the constructor whose parameters are of the same type as those with which the call was made.

### 4.3.3 The keyword ”this”

When we call a method that belongs to an object within the object, it is implicitly considered that the reference with which the method was called is the object that invokes it. This reference is ”this”. ”this” represents the current object instance. To better understand ”this”, we have the example below, where we calculate the power of a number:

```
1  class Power {
2      double b;
3      int e;
4      double val;
5
6      Power(double base, int exp) {
7          this.b = base;
8          this.e = exp;
9          this.val = 1;
10         if(exp == 0) return;
11         for(; exp > 0; exp--)
12             this.val = this.val * base;
13     }
14
15     double get_power() {
16         return this.val;
17     }
18 }
19
20 class DemoPower {
21     public static void main(String args[]) {
22         Power x = new Power(3, 2);
23         System.out.println(x.b + " to the " + x.e +
24                            " power is " + x.get_power());
25     }
26 }
```

In this case, in the constructor, the instruction ”this.b = base;” refers to the value ”b” declared as a member of the ”Power” class. When we create a new object, the constructor of ”Power” will

be called, and in the constructor we initialize the members of the new object, which is referred to as ”this”.

### 4.3.4 Method Overloading

In Java, it is possible to have two or more methods with the same name in the same class, as long as their parameters are different. This is called method overloading, and it is one of the ways that Java supports polymorphism.

What does it mean to have different parameters? There is a restriction: the parameters must be different - both the data type and the number of parameters can differ. If only the return type of the function is different, the compiler cannot choose which method to call. The return data types do not provide enough information to make a difference. In the following example, we will demonstrate method overloading:

```
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for one integer parameter.
  void test(int a) {
    System.out.println("a: " + a);
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }

  // Overload test for a double parameter
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}
class MethodOverload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    double result;

    // call all versions of test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Result of ob.test(123.25): " + result);
  }
}
```

In the above example, in the class OverloadDemo we have the method test overloaded three times after the first definition with no parameters:

```
void test(int a)
void test(int a, int b)
double test(double a)
```

As you can see, one of the overloaded methods returns a double value, while the others don't return anything (i.e., void). The test(double a) method will be called when test is called with a double value, and the other versions will be called with an integer value.

Method overloading is not only about changing the return data type of a function. Below is an example where we defined two functions with the same parameter but different return data types:

```
class NoOverloadDemo {
  void test(int a) {
    System.out.println("parametrul int: " + a);
  }

  // This will not compile!
  int test(int a) {
    System.out.println("a: " + a);
    return a*a;
  }
}

class MethodOverloadError {
  public static void main(String args[]) {
    NoOverloadDemo ob = new NoOverloadDemo();

    ob.test(7);
  }
}
```

In this example, we will get the following compilation errors:

```
NoOverloadDemo.java:8: test(int) is already defined in NoOverloadDemo
 int test(int a)
     ^
NoOverloadDemo.java:15: test(int) is already defined in NoOverloadDemo
  void test(int a) {
```

The test(int a) function was already defined with a void return type, so it cannot be redefined with a different return type.

Why do we need to overload functions? This mechanism is called the "one interface, multiple methods" paradigm, and in languages that do not support this concept, each method must have a different name.

## 4.4 Inheritance

Inheritance is a mechanism in object-oriented programming that allows a class to inherit properties and methods from another class. In Java, you can create a subclass by using the `extends` keyword. Here is an example:

```
public class SportsCar extends Car {
private boolean hasSpoiler;

public SportsCar(String make, String model, int year, boolean hasSpoiler) {
super(make, model, year);
this.hasSpoiler = hasSpoiler;
}

// additional methods go here
}

SportsCar mySportsCar = new SportsCar("Porsche", "911", 2022, true);
```

In this example, we have created a subclass of `Car` called `SportsCar`, which has an additional property (`hasSpoiler`). We have also created an object of this subclass called `mySportsCar`.

### 4.4.1 The role of constructors in inheritance

In a hierarchy resulting from inheritance, both superclasses and subclasses can have their own constructors. The question is, which constructor is responsible for instantiating the subclass object? Is it the one from the superclass, from the subclass, or both? The answer is the following: the superclass constructor will help instantiate the superclass portion of the object, and the subclass constructor will instantiate the subclass portion. To be more explicit, let's revisit the example with Shape2D and Triangle.

The superclass portion is automatically instantiated by calling the default constructor of the Forma2D class.

In addition to the above, a subclass can call the superclass constructor by using the keyword "super". The usage is as follows:

```
super(parameter list);
```

To see how this call is used, we will modify the class above. We defined a constructor in the Forma2D class with two double type parameters. In the constructor for Triangle, we replaced the two statements that initialized the members of Shape2D with a call to the parent class constructor.

This constructor:

```
Triangle(double a, double b, String tip) {
inaltime = a; // initializes the superclass portion
latime = b; // i.e. the Forma2D, superclass
type = tip; // instantiates the subclass portion
}
```

was replaced by this:

```
Triangle(double a, double b, String tip) {
super(a, b);
type = tip;
}
```

Here, the Triangle class calls the constructor of the Shape2D class with two double type parameters.

In the following example, we have demonstrated the use of superclass constructors with the keyword ”super”, including the use of zero-argument constructors and constructor overloading in the parent class:

```
public Shape2D() {
    height = 0; // in the zero-argument constructor
    width = 0; // members are initialized with zero
}

public Shape2D(double a, double b) {
    height = a;
    width = b;
}

void PrintDimension() {
    System.out.println("height is " +
    height + " width is " + width);
}
```

```
The "Triangle" class will have
class Triangle extends Shape2D {
String tip_triunghi;
Triangle(String tip) {
    super(); // call to Shape2D() and the superclass members will be zero
    type = tip;
}

Triangle(double a, double b, String tip) {
    super(a, b); // call to Shape2D(a, b)
    type = tip;
}

double CalcArea() {
    return height * width / 2;
}

void PrintType() {
    System.out.println("Traingle is " + type);
}
```

```
class DemoInheritance{
public static void main(String args[]) {
Triangle t1 = new Triangle("regular");
Triangle t2 = new Triangle(4, 6, "isosceles");
System.out.println("Info t1: ");
```

```
t1.PrintType();
t1.PrintDimension();
System.out.println("Area " + t1.CalcArea());
System.out.println();
System.out.println("Info t2: ");
t2.PrintType();
t2.PrintDimension();
System.out.println("Area " + t2.CalcArea());
System.out.println();
```

### 4.4.2 Method overriding

When a method in a child class has the same signature (parameters, name, and data type) as a method in the parent class, the method in the child class overrides the method in the parent class. When an overridden method is called from the child class, only the method in the child class is referenced (as if the method in the parent class does not exist).

```
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }
  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
class B extends A {
  int k;
  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
  // display only k
  void show() {
    System.out.println("k: " + k);
  }
}
class DemoOverride {
  public static void main(String args[]) {
    B obj = new B(1, 2, 3);
    obj.show(); // will call show() in B
  }
}
```

In the example above, class B is a child of class A. The show() method is overridden in B, which means it is redefined in B, even though it also exists in the parent class, A. In the DemoOverride class, when working with the object obj of type B, and calling the method obj.show(), it is as if class A does not define the show() method.

There may be confusion regarding the concept of overloading. Overloading implies that the same function has different parameters, which does not happen in the example above. However, we can implement an overloading of the show() method in B to illustrate the difference.

```
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }
  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
class B extends A {
  int k;
  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
  void show(String str) { //Overloaded show() method from A
    System.out.println(str);
  }
  // no more overridden show() method in class B
}
class DemoOverride {
  public static void main(String args[]) {
    B obj = new B(1, 2, 3);
    obj.show("Message 1"); //call show() from B
    obj.show(); // will call show() from A
  }
}
```

The reasons for maintaining this overriding mechanism are multiple, and it is one of the elements that contribute to polymorphism at runtime. Polymorphism is fundamental in OOP because it allows a general class to specify methods that will be the same for all derived classes from it, while allowing some of the child classes to have their own implementations for those methods.

When a method should not be overridden in any of the child classes, the `final` keyword can be added to that method. In the example given, if the `final` keyword is added to the `show()` method in class A, then if we try to override the final `show()` method in class B, we will receive an error message.

## 4.5 Object class

The Object class is a fundamental class in Java that all other classes directly or indirectly extend. It defines several important methods that can be overridden or used by other classes. These meth-

ods include clone(), equals(), finalize(), getClass(), hashCode(), toString(), wait(), and notify(). The getClass(), notify(), notifyAll(), and wait() methods are declared final. In the table below, the functions implemented in this class, are described.

| **Method** | **Purpose** |
|---|---|
| `Object clone()` | Create an object with the same properties as the cloned object |
| `boolean equals(Object object)` | Determines whether an object is/ is not equal to another |
| `void finalize()` | Called before the object is destroyed |
| `Class<? extends Object> getClass()` | etermines the class to which the object belongs |
| `int hashCode()` | Returns a specific ID for each object |
| `void notify()` | Resumes execution of the waiting execution thread |
| `void notifyAll()` | Resumes execution of all waiting execution threads |
| `String toString()` | Returns a string that describes the object |
| `void wait()` | Suspends the calling execution thread |

### 4.5.1 Overriding Methods

Many of these methods can be overridden to provide class-specific implementations. For example, you may override the `equals()` and `hashCode()` methods to provide your own equality checks and hash code calculations.

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }
    MyClass other = (MyClass) obj;
    return field1.equals(other.field1) && field2 == other.field2;
}

@Override
public int hashCode() {
    return Objects.hash(field1, field2);
}
```

### 4.5.2 The `Cloneable` Interface

For an object to be eligible for cloning, its class must implement the `Cloneable` interface, and you should override the `clone()` method. This will make more sense when after you will read

the next chapter about interfaces.

```
public class MyClass implements Cloneable {
    // ...
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

### 4.5.3 Concurrency Methods

The `wait()`, `notify()`, and `notifyAll()` methods are related to multi-threading and concurrency. They are used to synchronize activities of independently running threads in a program.

```
synchronized(myObject) {
    while(condition) {
        myObject.wait();
    }
    // Perform some action
    myObject.notifyAll();
}
```

By providing these functionalities, the *Object* class serves as a common ancestor, allowing objects of different types to be treated as objects of a common superclass. This is particularly useful for operations like object serialization, cloning, or comparison, which can be performed on objects of different types, as long as they adhere to certain contracts defined in the Object class.

## Exercises

### Exercise 1: Understanding ”this” Keyword

Explain the concept of the ”this” keyword in Java. Provide an example to illustrate how it is used within a class.

### Exercise 2: Using ”this” in Constructors

Consider the following Java class:

```
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Write a Java program that demonstrates the use of the ”this” keyword in constructors. Create an instance of the ‘Person‘ class and display the ‘name‘ and ‘age‘ attributes.

## Exercise 3: Inheritance in Java

Create a base class called ‘Vehicle‘ with attributes ‘make‘, ‘model‘, and ‘year‘. Provide a constructor to initialize these attributes. Then, create a subclass called ‘Car‘ that extends ‘Vehicle‘ and add an additional attribute ‘isConvertible‘.

Write a Java program that demonstrates inheritance by creating an instance of the ‘Car‘ class and initializing its attributes, including those inherited from the ‘Vehicle‘ class.

## Exercise 4: Using ”super” Keyword in Constructors

Enhance the ‘Vehicle‘ and ‘Car‘ classes from the previous exercise. Add constructors to both classes that take parameters to initialize their attributes. In the constructor of the ‘Car‘ class, use the ”super” keyword to call the constructor of the ‘Vehicle‘ class to initialize the inherited attributes.

Write a Java program that creates an instance of the ‘Car‘ class using the new constructors and displays the attributes.

## Exercise 5: Method Overloading in Inherited Classes

Create a method called ‘startEngine()‘ in both the ‘Vehicle‘ and ‘Car‘ classes. Overload this method in the ‘Car‘ class to provide a different behavior.

Write a Java program that demonstrates method overloading in inherited classes by creating instances of both ‘Vehicle‘ and ‘Car‘ and calling the ‘startEngine()‘ method on them.

## Exercise 6: Inheritance and ”this” Keyword

Modify the ‘Person‘ class from Exercise 2 to include an additional method called ‘displayInfo()‘ that displays the name and age of the person. Create a subclass called ‘Student‘ that extends ‘Person‘ and adds an attribute ‘studentId‘. In the ‘displayInfo()‘ method of the ‘Student‘ class, call the ‘displayInfo()‘ method of the ‘Person‘ class using the ”this” keyword.

Write a Java program that demonstrates this inheritance and method calling behavior by creating an instance of the ‘Student‘ class and displaying the student’s information.

## Exercise 7: Multiple Levels of Inheritance

Create a three-level hierarchy of classes. For example, you can have a base class ‘Animal‘, a subclass ‘Mammal‘ that extends ‘Animal‘, and a subclass ‘Dog‘ that extends ‘Mammal‘. Add appropriate attributes and methods at each level.

Write a Java program that demonstrates multiple levels of inheritance by creating an instance of the ‘Dog‘ class and accessing attributes and methods from all three classes.

# Chapter 5

# Advanced Object-Oriented Programming in Java

Advanced Object-Oriented Programming in Java builds on the foundation of basic OOP concepts and explores more complex concepts like abstract classes, interfaces, cloning objects.

## 5.1 Abstract classes

Let's take the example from the previous chapter where we derived two classes, Triangle and Rectangle, from the 2DShape class. Suppose we want to derive a larger number of classes, such as Rectangle, Triangle, Ellipse, Trapezium, and so on. We can observe that all these classes have two common elements, namely Surface and Perimeter or Circumference. To work easily with a set of objects that can be either Triangle or Ellipse, it is useful for them to belong to classes that have a superclass, namely 2DShape. In these conditions, 2DShape should implement the Surface and Perimeter functions, but the question is how if the shape is not a particular one, so we don't know exactly how these two methods will be implemented.

In these situations, Java allows the use of abstract methods.

An abstract method does not have a defined body, only a signature, and the function declaration ends with ";". Below are a series of rules for defining abstract methods and abstract classes:

1. Any class that contains an abstract method is automatically abstract, so it must be declared as such.

2. An abstract class cannot be instantiated.

3. A subclass of an abstract class can be instantiated only if it overrides the abstract methods of the parent and provides an implementation for all these methods. These child classes are also called concrete to emphasize that they are not abstract.

4. If a child class of an abstract superclass does not implement all the abstract methods it inherits, then the child class is also abstract and will be declared as such.

5. Static, private, or final declared methods cannot be abstract because none of these can be overridden by subclasses. Also, a class declared as final cannot contain abstract methods.

6. A class can be declared abstract even if it does not contain abstract methods. Usually, this signifies that a class is incomplete and is the parent class for one or more child classes.

Below is an example of using abstract classes:

```
public abstract class Shape {
    public String category;
    public abstract double area();
    public abstract double perimeter(); // only signature, no body
}

class Circle extends Shape {
    public static final double PI = 3.1415926535897;
    protected double radius;
    public Circle(double r) {
        this.radius = r;
        category = "Circle";
    }
    public double getRadius() {
        return radius;
    }
    // overriding the abstract methods
    public double area() {
        return PI * radius * radius;
    }
    public double perimeter() {
        return 2 * PI * radius;
    }
}
```

```
class Rectangle extends Shape {
    protected double width, height;
    public Rectangle(double width, double height) {
        category = "Rectangle";
        this.width = width;
        this.height = height;
    }
    public double getWidth() {
        return width;
    }
    public double getHeight() {
        return height;
    }
    // overriding the abstract methods
    public double area() {
        return width * height;
    }
    public double perimeter() {
        return 2 * (width + height);
```

```
    }
}
```

```
public class AbstractDemo {
    public static void main(String args[]) {
        // declare an array of abstract classes
        Shape[] shapes = new Shape[4];
        // instantiate various objects from the array
        shapes[0] = new Circle(2.0);
        shapes[1] = new Rectangle(1.0, 3.0);
        shapes[2] = new Rectangle(4.0, 2.0);
        shapes[3] = new Circle(5.0);
        double totalArea = 0;
        for(int i = 0; i < shapes.length; i++) {
            totalArea += shapes[i].area();
            // we can print these values because they are from the
            // parent class
            System.out.println("Category: " + shapes[i].category);
            System.out.println("Perimeter: " + shapes[i].perimeter());
            System.out.println("Area: " + shapes[i].area());
            // we cannot access these values because they are specific
            // to a subclass and not the parent
            // System.out.println("PI is: " + shapes[i].PI);
        }
        System.out.println("Total area: " + totalArea);
    }
}
```

In this example, we define an abstract class Shape, with two abstract methods area() and perimeter(). We also define two concrete classes Circle and Rectangle, which implement the Shape abstract class and provide their own implementation for the area() and perimeter() methods.

We then create an array of Shape objects and instantiate them with different Circle and Rectangle objects. We can call the area() and perimeter() methods on each object, and because they are from the parent class, we can print out their values.

However, we cannot access the PI constant from the Circle class, as it is specific to that subclass and not the parent class.

The `Shape` class is declared abstract and therefore cannot be instantiated. However, objects or arrays of objects are of type `Shape` because instantiation occurs later with a subclass of the parent, namely `Circle`:

```
sir_forme[0] = new Cerc(2.0);
```

Both the `Circle` and `Rectangle` classes are not abstract because they both implement all the abstract methods declared in the `Shape` class.

The two methods `Surface` and `Perimeter` are declared with the abstract keyword in the `Shape` class and have no body, so they will end with ";" like any normal declaration.

On the other hand, both child classes, `Circle` and `Rectangle`, implement both methods:

```
public double Surface()
```

```
{
    return PI *radius*radius;
}
public double Perimeter()
{
    return 2*PI*radius;
}
```

In addition to these two methods, both classes can contain their own methods separately. For example, Circle implements the following method:

```
public double getRadius()
{
    return r;
}
```

When using objects of these classes, we need to be careful because, although all objects in the array are of type Shape, the first and last object are of type Circle, while the second and third are of type Rectangle.

If an object is of type `Shape`, we cannot call a member (although public) of the `Circle` class. Such an instruction is incorrect:

```
System.out.println("PI is: " + sir[i].PI);
```

and produces the following compile error:

```
Shape.java: 86: cannot find symbol
symbol: variable PI
location: class Shape
System.out.println("PI is: " + sir[i].PI);
```

Furthermore, in the `Shape` parent class, there can be concrete methods in addition to abstract methods, such as this one:

```
public void ShowTypeOfShape()
{
    System.out.println("Shape is of type: " + category);
}

Example of using this function:

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

## 5.2 Interfaces

In OOP, it's recommended to define what a function should do without providing all the details on how to do those tasks. An abstract method is the answer to this problem. In some cases where

a class doesn't need concrete methods, interfaces can be created, which completely separate the implementation of a class from the definition of its methods.

Interfaces are similar to abstract classes, but in an interface, no method is allowed to have a body, i.e., implementation. Moreover, a class can implement one or more interfaces. For an interface implementation to be correct, a class must provide implementations for those methods declared in the interface. Obviously, each class can have its own implementation for interface methods.

The following is the general way of declaring an interface:

```
access interface interface_name {
data_type method_name1(param-list);
data_type method_name2(param-list);
data_type var1 = value;
data_type var2 = value;
// ...
data_type method_nameN(param-list);
data_type varN = value;
}
```

Where access is public or not used at all. The methods are declared using only their signature just like abstract methods. By default, the methods are public. Variables declared in an interface are not instance variables, but they must be declared public, final, and static, and must be initialized.

In the example above, the Shape class can implement an interface that can specify the center of a geometric shape:

```
1 public interface Centered {
2 void setCentre(double x, double y);
3     double getCentreX();
4     double getCentreY();
5 }
```

Also, an interface cannot be instantiated, and it cannot define a constructor.

### 5.2.1 Interface Inheriting

n the above code snippet, we are introduced to interfaces in Java. An interface defines a set of methods that a class must implement if it is to conform to that interface.

The syntax for declaring an interface is similar to that of declaring a class. It starts with the 'interface' keyword, followed by the name of the interface, and then a set of method signatures. The interface `Centered` declares three methods that a class must implement: setCentre, getCentreX, and getCentreY. Note that interfaces only define method signatures; they do not provide any implementation details.

In addition to defining methods, interfaces can also define constants. Constants declared in an interface are implicitly public, static, and final.

Interfaces can also extend other interfaces using the extends keyword. When one interface extends another interface, it inherits all the abstract methods of the parent interface, and can also declare additional abstract methods and constants. Here's an example:

```
public interface Positionable extends Centred {
    void setColRightUp(double x, double y);
    double getRightUpX();
    double getRightUpY();
}
```

In this example, the Positionable interface extends the Centrted interface and declares three additional methods.

Finally, an interface can also extend multiple interfaces, as shown in the example below:

```
public interface Transformable extends Scalable, Translateble, Rotateble {}
```

### 5.2.2 Interface Implementation

Just like a class uses extends to inherit from a parent class, it can use implements to inherit from interfaces. When a class inherits from an interface, it provides an implementation for all the interface methods. If a class implements an interface without providing an implementation for each method, it will inherit those abstract methods and become abstract itself. The general form of an implementation is:

```
access class class_name extends superclass implements interface {
// class body
}
```

Access is either public or not declared. The extends clause is optional, in case the original class extends other "regular" classes. What matters in the declaration above is the implements clause, which allows for the implementation of an interface. Below is an example of using interfaces:

```
abstract class Shape
{
    public String category;
    public abstract double Surface();
    public abstract double Perimeter(); // instead of the body, we have ";"
    public void ShowFormType()
    {
        System.out.println("The form is of type: " + category);
    }
}
```

```
class Rectangle extends Shape
{
protected double width, height;
public Rectangle(double width, double height)
{
    category = "Rectangle";
    this.width = width;
```

```
8       this.height = height;
9       }
10      public double getWidth()
11      {
12        return width;
13      }
14      public double getHeight()
15      {
16       return height;
17      }
18      // abstract methods that "capture the form":
19      public double Surface()
20      {
21        return width * height;
22      }
23      public double Perimeter()
24      {
25        return 2 * (width + height);
26      }
27  }
```

As can be seen, the declared interface is `Centered`, and the class that implements it is `CenteredRectangl` Since we are talking about a geometric shape of type rectangle, this class, `CenteredRectangle`, will also inherit from the `Rectangle` class.

```
1   public interface Centered
2   {
3       void setCenter(double x, double y);
4       double getCenterX();
5       double getCenterY();
6   }
7   class CenteredRectangle extends Rectangle implements Centered
8   {
9   // class fields
10  private double cx, cy;
11  // constructor
12  public CenteredRectangle(double cx, double cy, double width,
13  double height)
14  {
15      super(width, height);
16      this.cx = cx;
17      this.cy = cy;
18      }
19      // if we implement the Centered interface,
20      // we need to implement its methods
21      public void setCenter(double x, double y)
22      {
23       cx = x;
24       cy = y;
25      }
26      public double getCenterX()
27      {
```

```
28        return cx;
29    }
30    public double getCenterY()
31    {
32     return cy;
33    }
34 }
```

### 5.2.3 Interface and asbtract classes

When defining a data type for a class (e.g., Circle, Rectangle, etc.), we have to choose whether that data type is an abstract class or an interface. An interface is recommended because any class can implement it, even when that interface extends another superclass that has no connection to the interface. However, when an interface contains many methods, implementing all of them can become cumbersome for each class that implements it.

From this point of view, we can say that an abstract class does not have to be entirely abstract. It can contain at least partial implementation, which its subclasses can use. On the other hand, a class that extends an abstract class cannot extend (inherit) another class, which can lead to some difficulties.

Another difference between abstract classes and interfaces is related to compatibility. If we define an interface and add it to a library, and then add a method to that interface, the classes that implemented the previous version of the interface will be broken (they do not correctly implement the new interface). We say that we have broken compatibility. On the other hand, if we use abstract classes, we can add non-abstract methods without altering the classes that inherit it.

### 5.2.4 Interface and variables

Variables in interfaces must be static, public, and final. Basically, they are constants.

```
1  interface IConst
2  {
3      int MIN = 0;
4      int MAX;
5      String ERRORMSG = "Limit Error";
6  }
7  class DemoIConst implements IConst
8  {
9      public static void main(String args[])
10     {
11         int nums[] = new int[MAX];
12         for(int i=MIN; i < 11; i++)
13         {
14             if(i >= MAX) System.out.println(ERRORMSG);
15             else
16             {
17              nums[i] = i;
```

```
18                 System.out.print(nums[i] + " ");
19                 }
20             }
21         }
22 }
```

Any attempt to modify the value of a constant will fail at compile-time:

```
var.java:12: cannot assign a value to final variable MAX
MAX=20;
^
1 error
```

Also, any failure to meet the above conditions will result in various corresponding compilation errors.

In general, interfaces are useful when you want to define a contract that multiple classes can implement, while abstract classes are useful when you want to provide some common functionality to a group of related classes, but leave some methods to be implemented by the subclasses.

# Exercises

## Exercise 1: Using Interface in Class

Consider the class ‘Shape‘ that represents geometric figures. Apply the ‘IntStack‘ interface to this class to make it capable of storing and retrieving integers in a stack-like manner.

```
1 interface IntStack {
2     void push(int item); // store an item
3         int pop(); // retrieve an item
4         // Because clear( ) has a default, it need not be
5         // implemented by a preexisting class that uses IntStack.
6         default void clear() {
7         System.out.println("clear() not implemented.");
8     }
9 }
```

## Exercise 2: Implementing IntStack Interface

Create a class called ‘DynStack‘ that implements the ‘IntStack‘ interface. This class should have the following private attributes:

- ‘values‘: an array of integers to store stack elements.
- ‘top‘: an integer representing the top of the stack.

Implement the ‘push()‘ and ‘pop()‘ methods as specified in the ‘IntStack‘ interface. Additionally, implement a ‘toString()‘ method in the ‘DynStack‘ class to visualize the stack in the correct order.

## Exercise 3: Pushing Elements to the Stack

Using the ‘DynStack‘ class created in Exercise 2, add the following array to the stack: $\{2, 5, 3, 7, 6, 1\}$.

## Exercise 4: Displaying the Stack

Using the ‘toString()‘ method of the ‘DynStack‘ class, display the contents of the stack in the correct order.

## Exercise 5: IntQueue Interface and DynQueue Class

Create a new interface called ‘IntQueue‘ with the following methods:

- ‘enqueue(int item)‘: Add an item to the queue.
- ‘dequeue()‘: Remove and retrieve an item from the queue.
- ‘clear()‘: (Default method) Display a message indicating that the clear operation is not implemented.

Implement a class called ‘DynQueue‘ that implements the ‘IntQueue‘ interface. This class should allow simulation of a queue.

# Chapter 6

# Java Collections

Java provides a powerful and extensive set of data structures known as collections. Collections provide a way to store and manipulate groups of objects in a flexible and efficient manner: Naftalin (2007). In this chapter, we will explore the various types of collections available in Java and their associated methods and properties. A framework, in general, refers to a series of classes and libraries that act as a ”skeleton” in an application, allowing its expansion and development based on these elements. In Java, the Collections Framework is similar to the Standard Template Library (STL) in C++. There are about twenty-five classes and interfaces that constitute this core. This Collections Framework consists of three parts: interfaces, implementations, and algorithms. The implementations are the concrete classes that the framework provides, while the algorithms are predefined actions or methods that can exist within classes.

There are several interfaces in this framework, namely: Collection, List, Set, Map, SortedSet, and SortedMap. Their hierarchy is presented in the figure below:

In Figure 6.1 is depicted an overview of the main interfaces that are constructing the Collections Framework found in $java.util$ package.

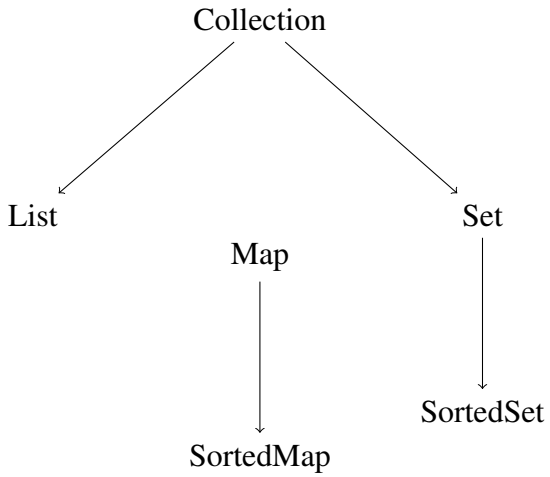



Figure 6.1: Interfaces in the Java Collections Framework

Below is a table showing the relationship between concrete classes and interfaces in the Java Collection Framework.

| | **HashTable** | **Mutable Array** | **Balanced Tree** | **Linked List** | **Others** |
|---|---|---|---|---|---|
| **Set** | HashSet | | TreeSet | | |
| **SortedSet** | | | TreeSet | | |
| **List** | | ArrayList | | LinkedList | Vector, Stack |
| **Map** | HashMap | | TreeMap | | Hashtable, Properties |
| **SortedMap** | | | TreeMap | | |

Table 6.1: Concrete classes inheriting from the interfaces in the Java Collection Framework

All classes either implement the `Collection` interface or the `Map` interface. When wanting to create classes implementing these interfaces, we should keep the following in mind:

- All implementations are unsynchronized. Synchronized access can be added, but it's not necessary.
- All classes provide *fail-fast* iterators. If a collection is modified while iterating through its elements, a `ConcurrentModificationException` will be thrown.
- All implementations work with `null` elements.
- The classes rely on a concept of optional methods in interfaces. If a class does not support a particular operation, it throws an `UnsupportedOperationException`.

# Collection Interface

This interface consists of a series of methods necessary for working with a collection, namely:

| **Method** | **Description** |
|---|---|
| `add()` | Adds an element to a collection |
| `addAll()` | Adds one collection to another collection |
| `clear()` | Removes all elements from a collection |
| `contains()` | Checks if an element is in a collection |
| `containsAll()` | Checks if one collection is entirely present in another collection |
| `equals()` | Checks the equality of two collections |
| `hashCode()` | Returns a unique ID specific to a collection |
| `isEmpty()` | Checks if a collection is empty |
| `iterator()` | Returns an object from a collection that allows the iteration through its elements |
| `remove()` | Removes an element from a collection |
| `removeAll()` | Removes elements of one collection from the current collection |
| `retainAll()` | Removes elements from a collection that aren't present in another collection |
| `size()` | Returns the number of elements in a collection |
| `toArray()` | Returns the elements of a collection as an array |

### Adding elements

You can add just one element using the `add` method. Normally, if no exception is thrown, this element will be in the collection after the function returns. In this case, the return value is true.

### Deleting elements

We can delete all elements from a collection using the `clear` method.

### Searching for elements

Before searching for an element, it is advisable to check its existence in the collection.

### Cloning collections

The Collection interface does not implement either Cloneable or Serializable. To copy a collection, it is advisable to pass it as a parameter to the object constructor when instantiating.

### Iterator interface

The iterator is that object that allows browsing collections. For this, it must implement the Iterator interface.

### Filter Iterator

Besides the possibility of browsing the elements from the collection, we can apply a predicate, namely the interface having a method that filters the elements from the collection.

More details regarding iterations and predicates will be treated in a next section of the book.

## 6.1 Generics

Generics allow you to write code that can work with a variety of different data types. This is useful because it allows you to write code that is more flexible and reusable. In the list below, there are a few benefits of using Generics:

- Stronger type checks at compile time: With generics, potential errors can be detected early, during compilation.
- Code reusability: A single generic class or method can be used with different data types.
- Elimination of type casts: Without generics, one often needs to use type casting, which can lead to runtime errors.

Generics are implemented using type parameters, which are specified in angle brackets (<>) after the name of the class or method. Here is an example:

```
public class Box<T> {
  private T content;

  public Box(T content) {
    this.content = content;
  }

  public T getContent() {
    return content;
  }

  public void setContent(T content) {
    this.content = content;
  }
}

Box<String> myBox = new Box<>("Hello, world!");
Box<Integer> myOtherBox = new Box<>(42);
```

The $Box < T >$ class definition illustrates a generic class. Here, $T$ is a type parameter. This $T$ can be replaced with an actual type when you instantiate the class.

Within the class, you can use $T$ as if it were a real type. For example, the content variable has the type $T$.

When creating instances of $Box$, you specify the type of content you want the box to hold. In the example, one box is defined to hold a String ($Box < String >$) and another box holds an Integer ($Box < Integer >$). More details about Generics will be treated further when we advance with this book.

In what follows we will review the most common types of concrete implementations of collections in Java, following in the next chapter a more detailed description of each of these collections.

## 6.2 ArrayList and LinkedList

An `ArrayList` is a resizable array implementation of the `List` interface. It allows for the storage of a variable number of objects and provides methods for adding, removing, and accessing elements. Here is an example of using an `ArrayList`:

```
ArrayList<String> myList = new ArrayList<>();
myList.add("apple");
myList.add("banana");
myList.add("cherry");
System.out.println(myList.get(1));
```

In this example, we have created an `ArrayList` of strings and added three elements to it. We have then printed the second element in the list using the `get()` method.

A `LinkedList` is a doubly-linked list implementation of the `List` interface. It allows for the storage of a variable number of objects and provides methods for adding, removing, and accessing elements. Here is an example of using a `LinkedList`:

```
LinkedList<String> myList = new LinkedList<>();
myList.add("apple");
myList.add("banana");
myList.add("cherry");
System.out.println(myList.get(1));
```

In this example, we have created a `LinkedList` of strings and added three elements to it. We have then printed the second element in the list using the `get()` method.

### 6.2.1 The differences between ArrayList and LinkedList

Both `ArrayList` and `LinkedList` are implementations of the `List` interface in Java's Collections Framework. However, they differ in several key aspects.

#### Internal Data Structure

- **ArrayList**: Uses a dynamic array to store elements.
- **LinkedList**: Uses a doubly-linked list data structure.

#### Performance

- **Random Access**:
  - `ArrayList`: Provides fast random access $O(1)$.
  - `LinkedList`: Provides slow random access $O(n)$, requiring traversal from the head to the required element.
- **Insertions and Deletions**:
  - `ArrayList`: Slower at insertions and deletions $O(n)$ due to potential array resizing and element shifting.
  - `LinkedList`: Faster at insertions and deletions $O(1)$ by only changing pointers.
- **Memory Overhead**:
  - `ArrayList`: Less memory overhead as it uses a single contiguous memory allocation.
  - `LinkedList`: Higher memory overhead due to the extra memory used by each node for two pointers (next and previous).

#### Capacity

- **ArrayList**: Size is dynamically increased but involves time-consuming array resizing and copying.
- **LinkedList**: No resizing required, more efficient at frequent insertions and deletions.

### 6.2.2 Null Elements

- Both `ArrayList` and `LinkedList` allow null elements.

#### Implementation

- **ArrayList**: A good general-purpose list for storage and random access.
- **LinkedList**: More specialized, advantageous for frequent insertions and deletions with infrequent random access.

### 6.2.3 Method Operations

- **ArrayList**: Provides basic methods like `add()`, `get()`, `remove()`, `size()`, etc.
- **LinkedList**: Offers additional methods for head and tail manipulation like `addFirst()`, `addLast()`, `pollFirst()`, `pollLast()`, etc.

#### Use-Cases

- **ArrayList**: Preferable for fast random access and relatively stable or infrequently changing datasets.
- **LinkedList**: Preferable for frequent insertions and deletions without the need for fast random access.

## 6.3 HashMap and HashSet

A `HashMap` is an implementation of the `Map` interface that stores key-value pairs. It provides methods for adding, removing, and accessing elements by their key. Here is an example of using a `HashMap`:

```
1 HashMap<String, Integer> myMap = new HashMap<>();
2 myMap.put("apple", 1);
3 myMap.put("banana", 2);
4 myMap.put("cherry", 3);
5 System.out.println(myMap.get("banana"));
```

In this example, we have created a `HashMap` of strings and integers and added three key-value pairs to it. We have then printed the value associated with the key "banana" using the `get()` method.

A `HashSet` is an implementation of the `Set` interface that stores unique elements. It provides methods for adding, removing, and accessing elements. Here is an example of using a `HashSet`:

```
HashSet<String> mySet = new HashSet<>();
mySet.add("apple");
mySet.add("banana");
mySet.add("cherry");
System.out.println(mySet.contains("banana"));
```

In this example, we have created a `HashSet` of strings and added three elements to it. We have then checked whether the set contains the element "banana" using the `contains()` method.

### 6.3.1 Difference Between HashMap and HashSet

Both `HashMap` and `HashSet` are part of Java's Collections Framework and are often used to store and manage collections of data. However, their underlying data structures and use-cases differ in several ways.

#### Fundamental Differences

- **HashMap**: Implements the `Map` interface and stores key-value pairs.
- **HashSet**: Implements the `Set` interface and stores unique elements only.

#### Internal Data Structure

- **HashMap**: Internally uses a hash table to store key-value pairs.
- **HashSet**: Internally uses a `HashMap` to store elements as keys and a constant (often `null`) as values.

#### Performance

- **Time Complexity**:
  - `HashMap`: Average time complexity for `get()` and `put()` operations is $O(1)$.
  - `HashSet`: Average time complexity for `add()`, `remove()`, and `contains()` operations is $O(1)$.

### Duplicated Entries

- **HashMap**: Allows duplicate values but not duplicate keys.
- **HashSet**: Does not allow duplicate elements.

### Null Entries

- **HashMap**: Allows one null key and multiple null values.
- **HashSet**: Allows one null element.

### Ordering

- **HashMap**: Does not maintain any order of keys or values.
- **HashSet**: Does not maintain any order of elements.

### Synchronization

- Both `HashMap` and `HashSet` are not synchronized, meaning they are not thread-safe by default.

### Use-Cases

- **HashMap**: Useful when we want to store and retrieve elements as key-value pairs, such as dictionaries.
- **HashSet**: Useful when we want to store a collection of unique elements and membership tests, such as a set of IDs.

## Equality in Lists

We have often stated that equality is checked using the `equals` method. This can be overridden when working with a specialized list. In addition to the `equals` method, the list item's IDs (i.e., what the `hashCode()` function returns) are also compared.

Consider the example below:

```
1 import java.util.ArrayList;
2 class Point {
3     public int x;
4     public int y;
5 
6     public Point (int x, int y) {
7         this.x = x;
8         this.y = y;
9     }
```

```
10
11      public boolean equals(Object o) {
12          if (!(o instanceof Point))
13              return false;
14          Point pt = (Point)o;
15          return ((pt.x == this.x) && (pt.y == this.y));
16      }
17
18      public int hashCode() {
19          return 17*this.x + 23*this.y + 43;
20      }
21
22      public String toString() {
23          return "x = " + x + " y = " + y + " id = " + hashCode() + "\n";
24      }
25  }
26
27  public class Mylist {
28      public static void main(String args[]) {
29          ArrayList list = new ArrayList();
30          list.add(new Point(1,2));
31          list.add(new Point(3,4));
32          list.add(new Point(2,1));
33          list.add(new String("a point"));
34          Point pt = new Point(-1,-1);
35          System.out.println(list.contains(pt));
36          System.out.println(list.contains("a point"));
37          System.out.println(list.contains(new Point(3,4)));
38          System.out.println(list);
39      }
40  }
```

The `hashcode()` function should always return a different number if the objects are different. We have:

```
list.add(new Point(1,2));
list.add(new Point(2,1));
```

If the `hashcode()` function returned the sum or the product of the two coordinates we would obtained the same result (3 for sum or 2 for product) for the two objects. This introduces a so called ”collision” issue, that means it does not respect principle of injectivity which is the main goal of the `hashcode()` function. A solution to avoid this issue is: multiplying by prime numbers the members values and then summing the products up. The program’s output is:

```
false
true
true
[x = 1 y = 2 id = 106 , x = 3 y = 4 id = 186 , x = 2 y = 1 id = 100 , a p
```

Obviously, the `String` object "a point" does not belong to the `Point` class. To ensure only certain data types are accepted in a list, generic lists are used.

In this chapter, we've explored various facets of collections in Java, including lists, sets, and maps. We delved into the nuances of different implementations such as `ArrayList`, `LinkedList`, `HashMap`, and `HashSet`, along with their respective advantages and limitations. Understanding the core differences between these collection types is crucial for writing efficient Java programs, as each type is better suited to specific use-cases.

In addition to basic types, Java's Collections Framework offers other advanced types and utilities, such as `TreeMap`, `PriorityQueue`, and `Collections` class methods for tasks like sorting and reversing. These tools further extend your ability to manipulate data structures effectively and we will study them in next chapters.

# Exercises

## Exercise 1: Word Frequency Analysis

Given the text:

> "Until recently, the prevailing view assumed lorem ipsum was born as a nonsense text. It's not Latin, though it looks like it, and it actually says nothing. Its words loosely approximate the frequency with which letters occur in English, which is why at a glance it looks pretty real."

Create a program that displays the words and their frequencies in the text. Use a 'Map' to store the word-frequency pairs.

## Exercise 2: Display Words in Reverse Alphabetical Order

Display the words and their frequencies in reverse alphabetical order.

## Exercise 3: Display Words by Frequency

Display the words and their frequencies in order of frequencies, from highest to lowest.

## Exercise 4: Create a Collection of All Words

Create a collection that stores all unique words encountered in the text, even if they appear multiple times.

## Exercise 5: Search for a Word

Implement a search function that allows users to find a word in the collection. For example, searching for "which" should return all positions (indices) where it was found in the list.

## Exercise 6: Sort the List Alphabetically

Sort the list of words in alphabetical order.

## Exercise 7: Generic Search

Implement a generic class ‘Searcher‘ that allows searching for the position of a given value in a generic array. Use the following method signature:

```
<T extends Comparable<T>> int search(T[] array, T key)
```

Handle exceptions that may occur during the search.

## Exercise 8: Pair Class Modification

Given the ‘Pair‘ class, modify it so that both members have the same type. Add a method ‘swap‘ to interchange the two members.

```
public class Pair<T, S>
{
    public Pair(T firstElement, S secondElement)
    {
        first = firstElement;
        second = secondElement;
    }
    public T getFirst() { return first; }
    public S getSecond() { return second; }
    private T first;
    private S second;
    }
}
```

## Exercise 9: Finding Min and Max

Add a static method ‘minmax‘ to the ‘PairUtil‘ class that calculates both the maximum and minimum values from an array of elements of type ‘T‘. This method should return a ‘Pair‘ object containing the maximum and minimum values.

# Chapter 7

# More Java Programming Concepts

Java is a powerful programming language with many advanced features that allow for the creation of complex, efficient applications. In this chapter, we will explore some of these advanced features and how they can be used to enhance your Java programming skills.

## 7.1 Exceptions in Java

Handling exceptions is a critical component of robust software development. Exceptions are anomalies that occur during the execution of a program. When these anomalies are not handled, they cause the program to terminate abruptly. Handling exceptions helps in gracefully managing these anomalies and ensuring that the program can continue running or terminate gracefully Oaks (2014). Exceptions are the way in which Java handles errors. They provide a robust mechanism to capture and respond to unexpected events during program execution.

### 7.1.1 Basic Exception Handling Structure

In Java, exceptions are dealt with using a `try-catch` mechanism. Here's the basic structure:

```
try {
   // block of code to monitor for errors
} catch (ExceptionType1 exOb) {
   // exception handler for ExceptionType1
} catch (ExceptionType2 exOb) {
   // exception handler for ExceptionType2
}
// ...
finally {
   // block of code to be executed after try block ends
}
```

Say the program starts, then calls Method1(), which in turn calls Method2(), and then an ArithmeticException is thrown. The direction of the arrows indicates the flow of control. The red arrow from Method2() to the ArithmeticException indicates where the exception is thrown.

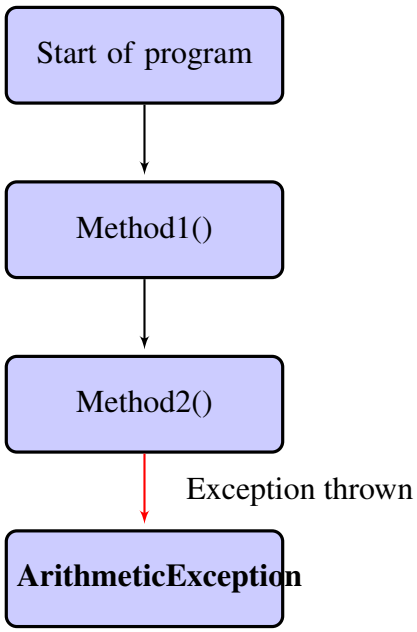



Figure 7.1: Representation of how exceptions are thrown in a stack trace.

If Method2() catches the exception, its said that the exception has been handled by this method. Otherwise the exception is passed to Method1() and so on. If no method catches the exception the program stops abruptly with the message error printed out to the output (usually the Console). In case the exception is handled by any of the methods, the program will continue with the instructions that follow the try catch block. Handlying

### 7.1.2 Hierarchy of Exceptions

Exceptions in Java are organized in a hierarchy:

- Throwable
  - Exception
  - Error
    - RuntimeException

### 7.1.3 Examples of Exceptions

In this section a few samples of throwing and catching exceptions will be found.

#### Basic Example

An example demonstrating an $ArithmeticException$ due to a division by zero:

```
class Exc2 {
   public static void main(String args[]) {
      int d, a;
      try {
         d = 0;
         a = 42 / d;
         System.out.println("This will not be printed.");
      } catch (ArithmeticException e) {
         System.out.println("Division by zero.");
      }
      System.out.println("After catch statement.");
   }
}
```

### Multiple `catch` Statements

You can have multiple catch blocks to handle different types of exceptions:

```
class MultipleCatches {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = {
                1
            };
            c[42] = 99;
        } catch (ArithmeticException e) {
            System.out.println("Divide by 0: " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index oob: " + e);
        }
        System.out.println("After try/catch blocks.");
    }
}
```

### Nested Exceptions

You can nest try-catch blocks inside other try-catch blocks:

```
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;
            /* If no command-line args are present,
            the following statement will generate
            a divide-by-zero exception. */
            int b = 42 / a;
            System.out.println("a = " + a);
```

```
            try { // nested try block
                /* If one command-line arg is used,
                then a divide-by-zero exception
                will be generated by the following code. */
                if (a == 1) a = a / (a - a); // division by zero
                /* If two command-line args are used,
                then generate an out-of-bounds exception. */
                if (a == 2) {
                    int c[] = {
                        1
                    };
                    c[42] = 99; // generate an out-of-bounds exception
                }
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index out-of-bounds: " + e);
            }
        } catch (ArithmeticException e) {
            System.out.println("Divide by 0: " + e);
        }
    }
}
```

### Throwing Exceptions

You can also throw exceptions using the `throw` keyword:

```
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch (NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }
    public static void main(String args[]) {
        try {
            demoproc();
        } catch (NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

### Declaring Exceptions

You can declare exceptions using the `throws` keyword:

```
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
```

```
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

You can create custom exceptions by extending the Exception class. Java allows exceptions to be chained, meaning one exception can be the cause of another. The initial exception serves as the root cause and can be retrieved using the method `getCause()`.

All good, but what to do when I catch an exception? Handling exceptions correctly is essential in order to maintain a stable application environment, inform the user properly, and preserve valuable diagnostic information for developers.

### 7.1.4 Strategies for Handling Exceptions

1. **Log the Exception**: Logging exceptions provides a trail of evidence for developers to diagnose the root causes of issues. Remember to:
   - Record the entire stack trace.
   - Provide context, if possible, about the ongoing operations when the exception occurred.
   - Ensure sensitive information is not logged.
2. **User Communication**: In many situations, you'll need to inform the user when an exception occurs, especially if it affects their interaction with the application. Ensure that:
   - The message is user-friendly and avoids technical jargon.
   - You provide guidance on the next steps, if applicable.
3. **Recovery or Graceful Degradation**: Where possible, try to recover from the exception or allow the application to continue running in a degraded mode.
4. **Rethrow or Chain Exceptions**: In some cases, you might want to catch a specific exception, do some processing or logging, and then throw another exception. This can be useful for abstraction layers, where a lower-level exception is translated to a higher-level one.
5. **Retry the Operation**: For transient failures (like a temporary network outage), you might consider retrying the operation. Implement mechanisms like exponential backoff to avoid hammering a system in rapid succession.

6. **Release Resources**: Ensure that any resources (like database connections, file handles, or network sockets) are released or closed to avoid resource leaks.

7. **Exit Gracefully**: In severe scenarios where recovery isn't possible, you might need to shut down the application. Ensure that this is done gracefully, releasing all resources and informing the user appropriately.

8. **External Reporting**: Consider using external error reporting tools or services which can capture, notify, and help in analyzing exceptions in real-time.

While it's impossible to anticipate every error, a robust error-handling strategy can mitigate the impact of unexpected issues and provide valuable insights for remediation. We will discuss a few aspects about one of the strategies, namely logging.

### 7.1.5 Logging

Logging is an essential mechanism in software development, allowing developers to record the internal operations of a program, monitor for specific events, and diagnose issues. Logging can track anything from informational messages and application flow to warnings, errors, and critical system failures.

#### Types of Logs

- **Debug Logs**: These provide detailed information about application execution, primarily used by developers to debug issues.
- **Info Logs**: General operational logs indicating the flow of the application.
- **Warning Logs**: These alert to something unexpected that might be an issue but doesn't interrupt the application's operation.
- **Error Logs**: Indicate when something has gone wrong, often including stack traces.
- **Fatal/Critical Logs**: These indicate severe issues that might cause the application to crash.

#### Common Logging Libraries

- **Log4j**: One of the most popular logging frameworks. Highly configurable through XML or properties files.
- **SLF4J**: Acts as a facade for different logging libraries.
- **java.util.logging**: Built into the Java SDK.
- **Logback**: Seen as the successor to Log4j.

**Logging Best Practices**

- Use appropriate **log levels**.
- Never log **sensitive information**.
- Implement **log rotation**.
- Use **centralized logging** systems.
- Ensure logging **redundancy**.
- For distributed systems, use **correlation IDs**.

Logging is an invaluable tool for monitoring, debugging, and ensuring the health of an application. By choosing the appropriate logging mechanism and adhering to best practices, developers can ensure they're well-prepared to tackle issues and understand the behavior of their applications.

## 7.2 Concurrency

Concurrency is the ability of a program to perform multiple tasks simultaneously. In Java, concurrency is achieved through the use of threads. A thread is a lightweight process that can run concurrently with other threads. Java provides several mechanisms for creating and managing threads, including the `Thread` class and the `Executor` framework.

### 7.2.1 Definition

A thread, defined as *sequence of instructions from a process executed in parallel with other similar sequences*, represents a single sequence of instructions within a process.

### 7.2.2 Difference from a Function

Unlike a simple function, a thread allows code to run in parallel with other sequences of instructions.

### 7.2.3 Thread Characteristics

- Threads can be synchronized.
- They can communicate through messages or by calling functions.

### 7.2.4 Main Thread

Every application has a main thread. Below is a demonstration of controlling the main thread:

```
// Controlling the main Thread.
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        t.setName("My Thread");
        System.out.println("After name change: " + t);
        try {
            for (int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

### 7.2.5 Creating a Thread

#### Method 1

Here's how a thread can be created using the *Runnable* interface:

```
class NewThread implements Runnable {
    Thread t;
    NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for the second thread.
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}
class ThreadDemo {
    public static void main(String args[]) {
```

```
24          new NewThread(); // create a new thread
25          try {
26              for (int i = 5; i > 0; i--) {
27                  System.out.println("Main Thread: " + i);
28                  Thread.sleep(1000);
29              }
30          } catch (InterruptedException e) {
31              System.out.println("Main thread interrupted.");
32          }
33          System.out.println("Main thread exiting.");
34      }
35  }
```

### Method 2

Another way to create a thread is by extending the *Thread* class:

```
1   class NewThread extends Thread {
2       NewThread() {
3           // Create a new, second thread
4           super("Demo Thread");
5           System.out.println("Child thread: " + this);
6           start(); // Start the thread
7       }
8       // This is the entry point for the second thread.
9       public void run() {
10          try {
11              for (int i = 5; i > 0; i--) {
12                  System.out.println("Child Thread: " +
13                      i);
14                  Thread.sleep(500);
15              }
16          } catch (InterruptedException e) {
17              System.out.println("Child interrupted.");
18          }
19          System.out.println("Exiting child thread.");
20      }
21  }
22  class ExtendThread {
23      public static void main(String args[]) {
24          new NewThread(); // create a new thread
25          try {
26              for (int i = 5; i > 0; i--) {
27                  System.out.println("Main Thread: " + i);
28                  Thread.sleep(1000);
29              }
30          } catch (InterruptedException e) {
31              System.out.println("Main thread interrupted.");
32              }
33              System.out.println("Main thread exiting.");
34          }
```

```
    }
```

### 7.2.6 Multiple Threads

It's possible to create and manage multiple threads in a single application:

```
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}
class MultiThreadDemo {
    public static void main(String args[]) {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");
        try {
            // wait for other threads to end
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

### 7.2.7 Using `isAlive` and `join`

The `isAlive()` method in the context of Java's Thread class is a fundamental method to ascertain the execution status of a thread. The importance and utility of isAlive() are manifold.

The primary purpose of `isAlive()` is to tell if a thread is still executing (running) or has terminated. This helps in monitoring and managing thread behavior.

When coordinating multiple threads, it is sometimes necessary to know if a particular thread has completed its task or is still running. For instance, a main thread might need to wait for other threads to finish before it continues, and checking the live status of those threads can be a determinant factor.

Before performing certain operations, it might be essential to ensure that a thread has completed its execution. This can prevent potential concurrency issues.

The `join()` method is an intrinsic component of Java's `Thread` class. Its main purpose is to allow one thread to wait until another completes its execution. Here's a deeper look into its significance:

1. **Sequential Execution**: In multi-threaded applications, there might be situations where it is necessary for one thread to complete its execution before another begins. The `join()` method provides a mechanism to achieve this sequential execution.

2. **Resource Dependence**: If one thread is producing some resource or result that another thread depends on, `join()` can ensure that the dependent thread waits for the producer thread to finish before proceeding.

3. **Ensuring Completion**: Before performing cleanup operations or exiting a program, it might be necessary to ensure that all spawned threads have completed. Using `join()` on these threads can achieve this.

4. **Avoiding Concurrency Issues**: In cases where multiple threads might access shared resources, using `join()` can help serialize access and avoid potential concurrency-related problems.

5. **Enhanced Debugging and Logging**: While debugging, the `join()` method can be used to observe the output of individual threads in a controlled sequence, making it easier to trace and debug issues.

This example demonstrates how to check if a thread is alive and how to wait for threads to complete their execution.

```
class DemoJoin {
    public static void main(String args[]) {
            NewThread ob1 = new NewThread("One");
            NewThread ob2 = new NewThread("Two");
            NewThread ob3 = new NewThread("Three");
            System.out.println("Thread One is alive: " +
                ob1.t.isAlive());
            System.out.println("Thread Two is alive: " +
                ob2.t.isAlive());
            System.out.println("Thread Three is alive: " +
                ob3.t.isAlive());
            // wait for threads to finish
            try {
                System.out.println("Waiting for threads to finish.");
```

```
15                      ob1.t.join(); ob2.t.join(); ob3.t.join();
16                  }
17                  catch (InterruptedException e) {
18                      System.out.println("Main thread Interrupted ");
19                      }
20              System.out.println("Thread One is alive: " +
21                      ob1.t.isAlive());
22              System.out.println("Thread Two is alive: " +
23                      ob2.t.isAlive());
24              System.out.println("Thread Three is alive: " +
25                      ob3.t.isAlive());
26              System.out.println("Main thread exiting.");
27          }
28      }
```

### 7.2.8 Synchronization

In multi-threaded programming environments, synchronization plays a pivotal role in ensuring data integrity, predictability, and efficient execution of tasks. Here are the primary reasons for its significance:

1. **Avoiding Race Conditions**: A race condition arises when two or more threads can access shared data and try to change it simultaneously. If the sequence in which the threads access the data affects the end result, then a race condition might occur. Synchronization ensures that only one thread can access the shared resource at a given time, thereby preventing race conditions.

2. **Data Integrity**: Without synchronization, threads might read outdated or incorrect values from shared resources. By synchronizing access, one can ensure that shared data remains consistent and uncorrupted.

3. **Ordered Execution**: Some processes need to happen in a specific sequence. Synchronization mechanisms can guarantee the order of execution where it's necessary.

4. **Deadlock Prevention**: Deadlock occurs when two or more threads are waiting indefinitely for a set of resources, with each thread holding a lock on one resource and waiting for another. Proper synchronization techniques, when implemented judiciously, can help prevent these scenarios.

5. **Efficient Resource Utilization**: By controlling access to shared resources, synchronization ensures that threads do not waste CPU cycles waiting for resources that are locked by other threads.

In essence, synchronization ensures that multi-threaded applications function predictably, securely, and efficiently. Without synchronization mechanisms, it would be challenging to develop reliable and scalable multi-threaded applications.

### Synchronization Method 1

The following example demonstrates a potential problem when multiple threads access shared resources.

```
class Callme {
    void call(String msg) {
        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        target.call(msg);
    }
}
class Synch {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}
```

### Synchronization Method 2

A safer method uses the ‘synchronized‘ keyword to ensure only one thread accesses shared resources at a time.

```
class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        synchronized(target) {
            target.call(msg);
        }
    }
}
```

### 7.2.9 Inter-Thread Communication

In multi-threaded applications, often there is a need for threads to communicate with each other. This communication is required to coordinate the execution of threads, especially when they operate on shared resources.

#### Importance of Inter-Thread Communication

- **Resource Sharing**: Threads might need to share resources like variables, data structures, or external entities. Proper communication ensures that the resources are used efficiently without conflicts.
- **Task Ordering**: Some tasks might need to be executed in a specific sequence, and inter-thread communication can ensure the correct order of task execution.
- **Efficiency**: Without proper inter-thread communication, threads might end up in busy-wait states, continually polling for a resource or condition. This can lead to wastage of CPU cycles.

## Mechanisms for Inter-Thread Communication

**Java's Built-in Methods** Java provides built-in mechanisms for inter-thread communication:

- `wait()`: This method is used by a thread to release the lock on the associated object and go into a wait state until another thread invokes `notify()` or `notifyAll()` on the same object.
- `notify()`: This method is used by a thread to wake up a single waiting thread on the associated object.

- `notifyAll()`: This method wakes up all the waiting threads on the associated object.

There are two main ways to control the communication between threads:

**Semaphores** These are synchronization tools that can control access to shared resources. A semaphore can signal and wait operations, allowing threads to communicate and coordinate their efforts.

**Message Passing** In some programming environments, threads communicate by sending and receiving messages. This method is especially prevalent in distributed systems where communication might occur between threads running on different machines.

In conclusion, inter-thread communication is essential to ensure the smooth and coordinated functioning of threads in a multi-threaded environment.

### Method 1

An incorrect implementation of producer and consumer, where the consumer might consume a value before the producer has produced it.

```
// An incorrect implementation of a producer and consumer.
class Q {
    int n;
    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }
    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}
class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while (true) {
            q.put(i++);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
```

```
}
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while (true) {
            q.get();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}
class PC {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
    }
}
```

### Method 2

A corrected implementation where the producer and consumer communicate to ensure proper order of operations.

```
class Q {
    int n;
    boolean valueSet = false;
    synchronized int get() {
            while (!valueSet) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    System.out.println("InterruptedException caught ");
                }
        }
        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }
    synchronized void put(int n) {
        while (valueSet) {
             try {
```

```
20                  wait();
21              } catch (InterruptedException e) {
22                  System.out.println("InterruptedException caught ");
23                  }
24          }
25          this.n = n;
26          valueSet = true;
27          System.out.println("Put: " + n);
28          notify();
29      }
30  }
```

In modern computing, multi-threading has emerged as a fundamental paradigm, pivotal for harnessing the capabilities of multi-core processors and for ensuring responsiveness in interactive applications. Threads, being the smallest unit of CPU execution, offer a more granular and efficient approach to multitasking compared to processes. They allow multiple sequences of instructions to execute concurrently within the context of a single process, sharing the same memory space.

The benefits of thread-based design are multi-fold:

- **Efficiency**: Threads have a lower overhead compared to processes. Their creation, destruction, and context switching are generally faster.
- **Responsiveness**: In interactive applications, multi-threading can ensure that the application remains responsive to user input, even when part of it is busy with intensive tasks.
- **Parallelism**: With the advent of multi-core processors, multi-threading allows software to leverage hardware parallelism, leading to significant performance gains for certain types of tasks.

However, with these advantages come challenges. Thread safety, synchronization, deadlock, race conditions, and inter-thread communication are complexities that developers must grapple with. Properly managed, threads can yield efficient and responsive software. But if mishandled, they can introduce elusive bugs and unpredictable behavior.

In the ever-evolving landscape of software development, understanding and mastering the principles of threading remain crucial for developers. As hardware continues its trend towards more cores and parallelism, the importance of threads in software design will only amplify.

## Thread Pools

In concurrent programming, managing the lifecycle of threads can be a challenging task, especially when dealing with a large number of threads. Starting and stopping threads can impose a significant overhead and might lead to system resource exhaustion. Here is where *thread pools* come into play.

### What is a Thread Pool?

A *thread pool* is a collection of worker threads that are initialized once and can be reused to execute multiple tasks. Instead of creating a new thread for every task, tasks are handed over to the threads available in the pool. Once a thread completes its task, it returns to the pool, ready to be assigned the next task.

### Benefits of Using Thread Pools

- **Resource Management**: Creating and destroying threads is costly in terms of time and system resources. By reusing existing threads, thread pools help in efficiently managing resources.
- **Improved Performance**: The overhead of thread creation is eliminated, leading to faster task startup.
- **Concurrency Control**: Thread pools allow developers to control the number of threads that are executing concurrently. This is particularly useful to avoid resource contention and potential system overloads.
- **Simpler Error Handling**: Handling errors in a pooled thread can be centralized, leading to more robust and maintainable code.

### Java's Executor Framework

In Java, the `java.util.concurrent` package provides the Executor framework, which abstracts thread pool management. The `ExecutorService` interface and its implementations, like `ThreadPoolExecutor` and `Executors`, provide out-of-the-box thread pool functionalities.

```
ExecutorService executor = Executors.newFixedThreadPool(10);
executor.submit(() -> {
    System.out.println("Task executed by: " + Thread.currentThread().getN
});
executor.shutdown();
```

In this simple example, a fixed thread pool with 10 threads is created. A task is then submitted to the pool for execution, and finally, the executor service is gracefully shut down.

**Note**: It's important to shut down the `ExecutorService` once its operations are done to free system resources and stop lingering threads.

## 7.3 Input and Output

This section covers input and output operations in Java, including reading and writing files and working with streams.

Java's `java.io` package offers classes and interfaces that are fundamental for file and stream operations. The 'File' class, for instance, represents file and directory pathnames in an abstract manner.

## Using the `File` Class

The 'File' class in Java is used for creating, reading, and managing files and directories.

```
class FileDemo {
    static void p(String s) {
        System.out.println(s);
    }
    public static void main
        (String args[]) {
            File f1 = new File("/java/COPYRIGHT");
            p("File Name: " + f1.getName());
            p("Path: " + f1.getPath());
            p("Abs Path: " + f1.getAbsolutePath());
            p("Parent: " + f1.getParent());
            p(f1.exists() ? "exists" : "does not exist");
            p(f1.canWrite() ? "is writeable" : "is not writeable");
            p(f1.canRead() ? "is readable" : "is not readable");
            p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
            p(f1.isFile() ? "is normal file" : "might be a named pipe");
            p(f1.isAbsolute() ? "is absolute" : "is not absolute");
            p("File last modified: " + f1.lastModified());
            p("File size: " + f1.length() + " Bytes");
        }
}
```

The output suggests the file's properties, like whether it's readable, writable, its size, and other metadata.

## Directories in Java

The 'File' class can also be used to work with directories:

```
String dirname = "/java";
File f1 = new File(dirname);
if (f1.isDirectory()) {
    System.out.println("Directory of " + dirname);
    String s[] = f1.list();
    for (int i = 0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
            System.out.println(s[i] + " is a directory");
        } else {
            System.out.println(s[i] + " is a file");
        }
    }
```

```
} else {
    System.out.println(dirname + " is not a directory");
}
```

## Using the `FilenameFilter` Interface

The `FilenameFilter` interface is used to filter filenames:

```
class OnlyExt implements FilenameFilter {
    String ext;
    public OnlyExt(String ext) {
        this.ext = "." + ext;
    }
    public boolean accept(File dir, String name) {
        return name.endsWith(ext);
    }
}
class FileDemo {
    static void p(String s) {
        System.out.println(s);
    }
    public static void main(String args[]) {
        String dirname = "/java";
        File f1 = new File(dirname);
        FilenameFilter only = new OnlyExt("html");
        String s[] = f1.list(only);
        for (int i = 0; i < s.length; i++) {
            System.out.println(s[i]);
        }
    }
}
```

This can be particularly useful when you want to filter out specific file types from a directory listing.

## Key Interfaces in `java.io`

- **AutoCloseable**: Has the method `void close() throws Exception`. It provides a standard mechanism to close objects.
- **Closeable**: Extends `AutoCloseable` and is specifically for objects that can be closed.
- **Flushable**: Has the method `void flush() throws IOException`. It flushes streams.

## Common I/O Exceptions

- `IOException`
- `FileNotFoundException`

- `SecurityException`

### Working with Streams

When working with any resource stream, including files and database connections, it's important to follow a standard pattern:

```
try {
    // open and access resource
} catch (Exception e) {
    // handle exception
} finally {
    // close the resource
}
```

Note: It's essential to close resources to free up system resources and prevent potential memory leaks or other unexpected behaviors.

## 7.3.1 Input from Console

This code snippet reads individual characters from the console until the user inputs the 'q' character. It uses a `BufferedReader` combined with an `InputStreamReader` that takes `System.in` as its argument:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
char c;
do {
    c = (char) br.read();
    System.out.println(c);
} while (c != 'q');
```

The `br.read()` method reads a single character, which is then type-casted into a `char` type and printed to the console.

## 7.3.2 Reading a String from the Console

This snippet captures entire lines (strings) from the console. The reading continues until the user enters the string "stop":

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str;
do {
    str = br.readLine();
    System.out.println(str);
} while (!str.equals("stop"));
```

The `br.readLine()` method reads an entire line of text, which is then printed to the console.

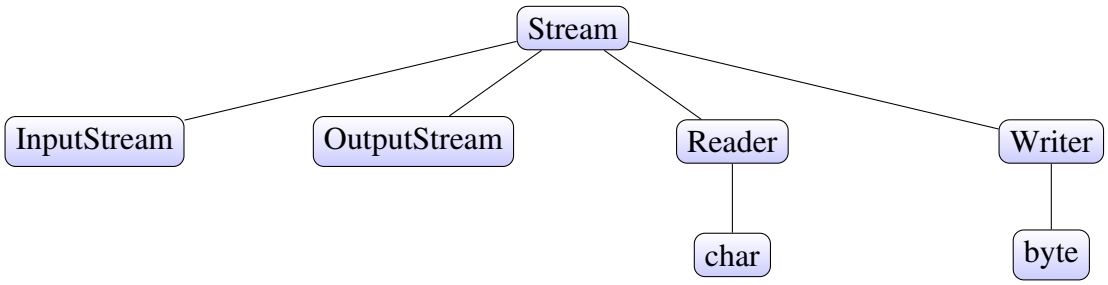



Figure 7.2: Representation of the Stream classes hierarchy.

### 7.3.3 Writing to the Console

This snippet demonstrates two ways to write to the console:

```
int b = 'A';
System.out.write(b);
PrintWriter pw = new PrintWriter(System.out, true);
pw.println("This is a string");
pw.println(-7);
pw.println(4.5e-7);
```

The `PrintWriter` is initialized with `System.out` as the output stream and `true` for auto-flushing the output buffer. This means after each `println` invocation, the data is immediately written to the console.

### 7.3.4 Stream Classes

## Java I/O Stream Classes

Java's I/O classes are organized into streams. Streams are sequences of data that provide a consistent way to read from and write to data sources, such as files or the console. The Java I/O classes from Figure 7.2 are structured around the key abstract classes from the `java.io` package: `InputStream`, `OutputStream`, `Reader`, and `Writer`.

#### InputStream and OutputStream: Byte Streams

- **InputStream**: The abstract superclass of all classes representing an input stream of bytes. Commonly used byte streams are `FileInputStream` (reading bytes from a file) and `BufferedInputStream` (reading from another input stream with buffering).
  - *Key methods*: `read()`, `available()`, `close()`
- **OutputStream**: The abstract superclass representing an output stream of bytes. Frequently used byte output streams are `FileOutputStream` (writing bytes to a file) and `BufferedOutputStream` (writing to another output stream with buffering).
  - *Key methods*: `write()`, `flush()`, `close()`

### Reader and Writer: Character Streams

- **Reader**: The abstract superclass for reading character streams. Notable subclasses are `FileReader` (reading characters from a file), `BufferedReader` (reading characters from another Reader with buffering), and `InputStreamReader` (converting byte streams to character streams).
  - *Key methods*: `read()`, `mark()`, `reset()`, `close()`
- **Writer**: The abstract superclass for writing to character streams. Prominent subclasses include `FileWriter` (writing characters to a file), `BufferedWriter` (writing characters to another Writer with buffering), and `OutputStreamWriter` (converting character streams to byte streams).
  - *Key methods*: `write()`, `flush()`, `close()`

### Additional Notes

- **Byte Streams vs. Character Streams**: The primary distinction is that byte streams handle raw binary data, while character streams handle text data. Given that text data is character-based and might need character encoding conversions, using character streams is often more appropriate for text files.
- **Buffering**: Buffering can enhance the performance of I/O. Without it, each read or write method call would cause bytes to be read from or written to a file, which can be time-consuming. By using classes like `BufferedInputStream` or `BufferedWriter`, the Java platform can efficiently read or write large chunks, minimizing costly operations.

### BufferedReader

Reading from a file using a ‘BufferedReader‘:

```
try {
    File f = new File("file2.txt");
    BufferedReader b = new BufferedReader(new FileReader(f));
    String readLine = "";
    System.out.println("Reading file using Buffered Reader");
    while ((readLine = b.readLine()) != null) {
        System.out.println(readLine);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

#### BufferedWriter

Writing to a file using ‘BufferedWriter‘:

```
try {
    File fout = new File("out.txt");
    FileOutputStream fos = new FileOutputStream(fout);
    BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));
    for (int i = 0; i < 10; i++) {
        bw.write("line no " + i);
        bw.newLine();
    }
    bw.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

### 7.3.5 Serialization

Serialization is the process by which an object's state is converted into a byte stream. Conversely, deserialization is the process by which a byte stream is used to recreate an object. Java provides a mechanism to persist object states and to transport them over the network through its `Serializable` interface.

#### Key Concepts

- **Serializable Interface**: Java objects that are meant to be serialized must implement the `Serializable` interface, which serves as a marker interface and doesn't define any methods.
- **ObjectOutputStream & ObjectInputStream**: These classes from the `java.io` package provide the actual mechanism for serializing and deserializing objects. Common methods include `writeObject()` for serialization and `readObject()` for deserialization.
- **transient Keyword**: Fields declared as `transient` are not serialized. This is useful when a field is derived from other fields or when it shouldn't be persisted for security or other reasons.

#### Advantages

- **Persistence**: Serialized objects can be saved to disk, allowing for object persistence across sessions.
- **Network Communication**: Serialization allows objects to be sent over a network, facilitating remote method invocation and other forms of network communication.
- **Deep Copy**: Serialization can be used to create a deep copy of an object. By serializing and then immediately deserializing an object, a new object with the same state is produced.

Serializing and deserializing an object:

```
class MyClass implements Serializable {
    String s;
    int i;
    double d;

    public MyClass(String s, int i, double d) {
        this.s = s;
        this.i = i;
        this.d = d;
    }

    public String toString() {
        return "s=" + s + "; i=" + i + "; d=" + d;
    }
}

// Object serialization
try (ObjectOutputStream objOStrm = new ObjectOutputStream
    (new FileOutputStream("serial"))) {
    MyClass object1 = new MyClass("Hello", -7, 2.7e10);
    MyClass object2 = new MyClass("Some String", 18, 2.34);
    objOStrm.writeObject(object1);
    objOStrm.writeObject(object2);
} catch (IOException e) {
    System.out.println("Exception during serialization: " + e);
}

// Object deserialization
try (ObjectInputStream objIStrm = new ObjectInputStream
    (new FileInputStream("serial"))) {
    MyClass object1 = (MyClass) objIStrm.readObject();
    MyClass object2 = (MyClass) objIStrm.readObject();
    System.out.println("object1 : " + object1);
    System.out.println("object2 : " + object2);
} catch (Exception e) {
    System.out.println("Exception during deserialization: " + e);
}
```

It's important to note that while serialization is powerful, it comes with security and performance considerations. Developers should be cautious when deserializing objects from untrusted sources.

## 7.4 Lambda Expressions

Introduced in Java 8, lambda expressions provide a concise way to represent functional interfaces. A functional interface is an interface that contains just one abstract method (though it may contain additional default or static methods). Lambda expressions allow you to express instances of single-method interfaces in a much more concise and expressive manner.

## Basic Syntax

A lambda expression is characterized by the following syntax:

```
(parameters) -> expression
```

or

```
(parameters) -> { statements; }
```

## Examples

- **No Parameters**

```
() -> System.out.println("Hello World")
```

- **Single Parameter (without type)**

```
name -> System.out.println("Hello, " + name)
```

- **Multiple Parameters**

```
(x, y) -> x + y
```

- **Multiple Statements in Body**

```
(x, y) -> {
   int result = x + y;
   System.out.println("Result: " + result);
   return result;
}
```

## Usage with Functional Interfaces

Lambda expressions are often used with predefined functional interfaces in Java, like ‘Predicate‘, ‘Function‘, and ‘Consumer‘.

**Example with ‘Predicate‘ interface**:

```
import java.util.function.Predicate;

public class LambdaDemo {
    public static void main(String[] args) {
        Predicate<Integer> isEven = number -> number % 2 == 0;
        System.out.println(isEven.test(10));  // Outputs: true
    }
}
```

### 7.4.1 Examples

When combined with the Stream API, the capabilities of lambda expressions truly shine. Here's a demonstration of how lambdas can be employed with a list of integers:

```
import java.util.Arrays;
import java.util.List;

public class LambdaWithStreams {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Leveraging lambda to sieve even numbers, square them, and then print
        numbers.stream()
                .filter(n -> n % 2 == 0)
                .map(n -> n * n)
                .forEach(System.out::println);
    }
}
```

In the example above, a lambda expression is being used to filter even numbers from a list, square them, and then print them. It manages to be concise, transparent, and declarative, clarifying the intent behind the code. You can incorporate this into your LaTeX document for further processing.

### Other Benefits

Lambda expressions provide several advantages:

- **Conciseness**: Reduces boilerplate code.
- **Functional Programming**: Introduces functional programming capabilities in Java.
- **Readability**: Enhances code readability by focusing on business logic.
- **Use with Streams**: Stream API in Java 8 works seamlessly with lambda expressions.

In this section, we'll explore how to read from a file in parallel using lambda expressions and a custom class in Java.

### Example: Using custom class for Parallel Processing

We'll employ the 'MyClass' to manage threads and process the lines of the file in parallel.

```
import java.io.IOException;
import java.nio.file.*;
import java.util.List;
import java.util.concurrent.*;
```

```
public class ParallelFileReader {

    public static class MyClass {
        private final ExecutorService executor;

        public MyClass(int numberOfThreads) {
            this.executor = Executors.newFixedThreadPool(numberOfThreads);
        }

        public void processLine(String line) {
            executor.submit(() -> {
                // Processing each line here
                System.out.println(Thread.currentThread().getName() + ": " +
    line);
            });
        }

        public void shutdown() {
            executor.shutdown();
        }
    }

    public static void main(String[] args) {
        Path path = Paths.get("a.txt");
        MyClass myClass = new MyClass(4); // Using 4 threads for
   demonstration

        try {
            List < String > lines = Files.readAllLines(path);
            lines.forEach(myClass::processLine);

            // Shutdown the executor service after processing all lines
            myClass.shutdown();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In this example, the ‘MyClass‘ is responsible for managing the parallelism. It uses an ‘ExecutorService‘ to submit tasks, which in this case are lambda expressions that process the lines from the file. Each line from the file is processed by a separate thread managed by the ‘ExecutorService‘.

# Exercises

## Exercise 1: Generating Permutations

Implement a method to generate permutations of a generic array with ‘n‘ elements. The value of ‘n‘ should be read from the keyboard. Ensure that the method handles exceptions gracefully.

## Exercise 2: Exception Handling

In all of exercises, handle exceptions appropriately. Ensure that any exceptions that might occur are caught and handled gracefully.

## Exercise 3: CSV File Handling

- Create a class ‘Point‘ with integer members ‘x‘, ‘y‘, and ‘z‘.
- Save the collection of points [(2,3,7), (-1,3,4), (5,4,0), (4,-2,6)] to a CSV (comma-separated values) file.
- Read the CSV file in another program and display the points.
- Append a new point (1,4,1) to the end of the file.
- Insert a new point (3,-6,2) into the file after a specific point (e.g., after (5,4,0)).
- Repeat all the above requirements using binary file handling (serialization).
- Extra: Repeat all the above requirements using XML or JSON file handling.

## Exercise 4: Multithreading

I Perform the sum of the first 10 natural numbers on a separate thread from ‘Main‘.

II Given the sequence [3,4,6,7,2,8,10,-1,3,-4,5,6,-2,-8,3,5,7,3,2,1], use two separate threads to calculate the sum of elements in the following way:

a. The first thread calculates the sum up to n/2.
b. The second thread calculates the sum of elements [n/2, n].
c. Each thread requests the next element from the sequence, so the sum will be distributed almost alternately.
d. Thread 1 calculates the sum of elements at odd positions, while thread 2 calculates the sum of elements at even positions.

III Read the content of a file on a separate thread from ‘Main‘.

IV Write a string to a file on a separate thread from ‘Main‘.

# Chapter 8

# A closer look to Java Collections

The Java Collections Framework, often simply referred to as Java Collections, stands as one of the cornerstones of the Java programming language. It provides a comprehensive suite of data structures and algorithms that empower developers to handle data efficiently and in a structured manner. Whether you're designing a complex data-driven application or simply organizing a list of items, Java Collections offers a solution tailored to your needs.

In this chapter, we'll dive deeper into the intricacies of Java Collections. We will explore the nuances of different Collection types, understand their optimal use cases, and get acquainted with the methods and functionalities they offer. By the end of this chapter, you'll have a robust understanding of the Collections Framework and will be equipped to leverage its full potential in your Java projects.

## 8.1 TreeSet

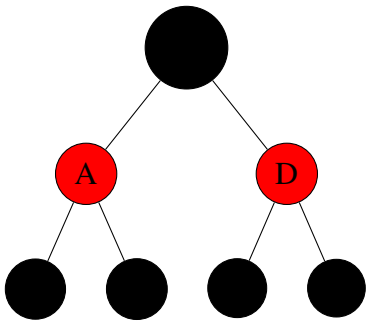
The `TreeSet` class functions similarly to a `HashSet`, but it maintains the elements in an order. These elements are ordered in a balanced tree, specifically, a red-black tree. By keeping the elements in a red-black tree, the cost of a search becomes logarithmic with a complexity order of $O(\log n)$.

### 8.1.1 Red-Black Tree Properties

A red-black tree adheres to the following rules:

1. Each node is red or black.
2. The root is always a black node.
3. If the node is red, its children must be black.
4. Every path from the root to leaves must contain the same number of black nodes.

An example of red and black tree is presented in the below structure.




### 8.1.2 TreeSet Methods

Below are some of the methods implemented by this class:

| Method | Description |
|---|---|
| `TreeSet()` | Constructor of a TreeSet. |
| `add()` | Adds an element to the set. |
| `addAll()` | Adds a collection of elements to the set. |
| `clear()` | Removes all elements from the set. |
| `clone()` | Creates a clone of the set with its elements. |
| `comparator()` | Returns a Comparator object. |
| `contains()` | Checks if an object exists in the set. |
| `first()` | Returns the first element of the set. |
| `headSet()` | Returns a subset of elements from the beginning of the set. |
| `isEmpty()` | Checks if the set is empty. |
| `iterator()` | Returns an iterator to iterate over the set. |
| `last()` | Returns the last element of the set. |
| `remove()` | Removes an element from the set. |
| `size()` | Returns the number of elements in the set. |
| `subSet()` | Returns a subset of the initial set. |
| `tailSet()` | Returns a subset of elements from the end of the initial set. |

Table 8.1: Methods in the `TreeSet` Class

### 8.1.3 TreeSet Examples

Here 8.1 is an example of how to use the Treeset to store a collection in an ordered manner. A TreeSet is created with a custom comparator that orders the elements in reverse. The Collections.reverseOrder() returns a comparator that imposes the reverse of the natural ordering on a collection of objects that implement the Comparable interface. The code uses a for loop to iterate through each string in the elements array and adds them to the TreeSet.

Listing 8.1: My Java code

```
import java.util.*;
public class Comparare {
    public static void main (String args[]) throws Exception {
        String elements[] = {"Radu", "Andrei", "Ion","Vasile", "Mircea"};
        // Creation of a set with a comparator
        Set set = new TreeSet(Collections.reverseOrder());
        for (int i=0, n=elements.length; i<n; i++) {
            set.add(elements[i]);
        }
        // Print the elements of the set
        System.out.println(set);

        // Print the comparator of the current set
        System.out.println(((TreeSet)set).comparator());
    }
}
```

The last line prints the comparator used by the TreeSet. Since a reverse comparator was used, this will print the class name of that comparator.

## 8.2 Sorting Collections

In this section, we will see which mechanism collections implement to sort elements as efficiently as possible. There are essentially two ways to sort collections: by implementing the `Comparable` interface or through a custom `Comparator`.

### 8.2.1 The Comparable Interface

Java-defined classes already implement the `Comparable` interface. This interface has one method, namely `compareTo()`, which defines how objects are compared and thus sorted. Below, we have a series of classes that naturally implement this interface.

#### Conditions for the compareTo method

- Elements must be mutually comparable.
- The return value expresses relative position in a natural order.
- Natural ordering should be based on the `equals()` method.
- Never call the method directly.

```
import java.util.*;

public class Employee implements Comparable {
    String department;
```

| Class Name | Arrangement |
|---|---|
| BigDecimal | Numeric (signed) |
| BigInteger | Numeric (signed) |
| Byte | Numeric (signed) |
| Character | Numeric (unsigned) |
| CollationKey | Alphabetically, based on local settings |
| Date | Chronological |
| Double | Numeric (signed) |
| File | Alphabetically by path |
| Float | Numeric (signed) |
| Integer | Numeric (signed) |
| Long | Numeric (signed) |
| ObjectStreamField | Alphabetically as a String |
| Short | Numeric (signed) |
| String | Alphabetical |

Table 8.2: Java classes implementing the Comparable interface.

```
5     String name;
6
7     public Employee(String department, String name) {
8         this.department = department;
9         this.name = name;
10    }
11
12    public String getDepartment() {
13        return department;
14    }
15
16    public String getName() {
17        return name;
18    }
19
20    @Override
21    public String toString() {
22        return "[dept=" + department + ", name=" + name + "]";
23    }
24
25    @Override
26    public int compareTo(Object obj) {
27        Employee toCompare = (Employee) obj;
28        int deptComp = department.compareTo(toCompare.getDepartment());
29        return (deptComp == 0) ? name.compareTo(toCompare.getName()) :
    deptComp;
30    }
31
32    @Override
33    public boolean equals(Object obj) {
```

```
34          if (!(obj instanceof Employee)) {
35              return false;
36          }
37          Employee e = (Employee) obj;
38          return department.equals(e.getDepartment()) && name.equals(e.getName
        ());
39      }
40
41      @Override
42      public int hashCode() {
43          return 43 * department.hashCode() + name.hashCode();
44      }
45  }
```

### 8.2.2 Comparator

When we don't want to impose the natural ordering of classes, or the classes do not implement the `Comparable` interface, we have the `compare` method from the `Comparator` interface available.

#### Using the Comparator

For the scenario when a new manager inspects the company and wishes to know an employee's name before their department, we need to implement a separate class from the base class, namely a Comparator.

```
1  class EmployeeComparator implements Comparator {
2      @Override
3      public int compare(Object obj1, Object obj2) {
4          Employee e1 = (Employee) obj1;
5          Employee e2 = (Employee) obj2;
6          int nameComp = e1.getName().compareTo(e2.getName());
7          return (nameComp == 0) ? e1.getDepartment().compareTo(e2.
       getDepartment()) : nameComp;
8      }
9  }
```

This class can act as a comparator for a new list, as we'll see below. This time, the `compare` method has two objects, `obj1` and `obj2`, which it will use for comparison. The logic is reversed from the previous example. It first checks the employee's name and then verifies the name of the department to which he belongs. The use of this Comparator is in the rewritten `main` function:

```
1  public static void main (String[] args) {
2      Employee employees[] = {
3          new Employee("HR", "Irina"),
4          new Employee("HR", "Cristina"),
5          new Employee("Engineer", "Daniel"),
6          new Employee("Engineer", "Octavian"),
7          new Employee("Sales", "Emil"),
```

```
        new Employee("RD", "Bogdan")
    };
    // The new collection has a comparator of type EmployeeComparator
    Set set = new TreeSet(new EmployeeComparator());
    set.addAll(Arrays.asList(employees));
    // We go through the already sorted set
    // but it contains objects
    for(Object o : set) {
        // So we'll need to cast
        Employee emp = (Employee) o;
        System.out.println(emp);
    }
}
```

Next, we will discuss collections that contain not just one element, but on each position, there are two elements, that is, a pair made up of a key and a value.

## 8.3 Dictionary, HashTable and Properties

Dictionary, HashTable, and Properties classes are fundamental in handling key-value pairs for various applications.

Dictionary is an abstract class containing only abstract methods. It can be visualized analogously to a phone book, where the person's name acts as a unique key and the phone number acts as the value. Notable methods in the Dictionary class include:

- `elements()`
- `get()`
- `isEmpty()`
- `keys()`
- `put()`
- `remove()`
- `size()`

### 8.3.1 HashTable Class

HashTable is a concrete implementation of the Dictionary class, which uses a hashing algorithm to provide swift lookups. The hashing function converts keys into hash codes, which accelerate searches within the collection.

### Notable Methods

| Method | Description |
|---|---|
| `clear()` | Removes all elements from the HashTable. |
| `clone()` | Creates a shallow copy of the HashTable. |
| `contains()` | Checks if an object is present in the HashTable. |
| `containsKey()` | Checks if a specific key exists. |
| `containsValue()` | Checks if a specific value exists. |
| `get(Object key)` | Retrieves the value associated with the given key. |
| `hashCode()` | Computes the hash code of the HashTable. |
| `isEmpty()` | Checks if the HashTable is empty. |
| `keys()` | Returns an enumeration of the keys. |
| `put(Object key, Object value)` | Associates a value with a key. |
| `remove(Object key)` | Removes the key-value pair associated with the key. |
| `size()` | Returns the number of key-value pairs. |

### An example

HashTable insertion operations generally work in constant time, O(1), due to its hash-based implementation. However, hash collisions can potentially reduce the effectiveness of the HashTable.

Listing 8.2: Java code for counting word occurrences in a text file

```
import java.io.*;
import java.util.*;

public class Cuvinte {
    static final Integer ONE = new Integer(1);

    public static void main (String args[]) throws IOException {
        Hashtable map = new Hashtable();
        FileReader fr = new FileReader(args[0]);
        BufferedReader br = new BufferedReader(fr);
        String line;
        while ((line = br.readLine()) != null) {
            processLine(line, map);
        }
        Enumeration en = map.keys();
        while (en.hasMoreElements()) {
            String key = (String)en.nextElement();
            System.out.println(key + " : " + map.get(key));
        }
    }

    static void processLine(String line, Map map) {
        StringTokenizer st = new StringTokenizer(line);
        while (st.hasMoreTokens()) {
```

```
            addWord(map, st.nextToken());
        }
    }

    static void addWord(Map map, String word) {
        Object obj = map.get(word);
        if (obj == null) {
            map.put(word, ONE);
        } else {
            int i = ((Integer)obj).intValue() + 1;
            map.put(word, new Integer(i));
        }
    }
}
```

The idea of this program is that, within the map (which is of type HashTable), we maintain a key-value pair, where the key represents the word read from the file, and the value represents the number of occurrences of that word.

The method processLine() divides the read line into words separated by spaces and then calls the addWord method. Here, it checks if the word has already been recorded in the collection. If not, it adds the pair (new word, 1) indicating that the word appears once. Otherwise, if it has been recorded before, the value corresponding to the key is incremented by one.

### 8.3.2 Properties Class

This section will provide insights into the `Properties` class and its unique methods.

The Properties class represents a specialized `HashTable`. In this class, both the keys and the values are of type String. When working with both `HashTable` and `Properties`, we need to be mindful of the hierarchy: a `Properties` can be a `HashTable`, but a `HashTable` cannot function as a Properties.

Consider a scenario where you want to store application settings:

```
# settings.properties
username=admin
password=mysecret
theme=dark
```

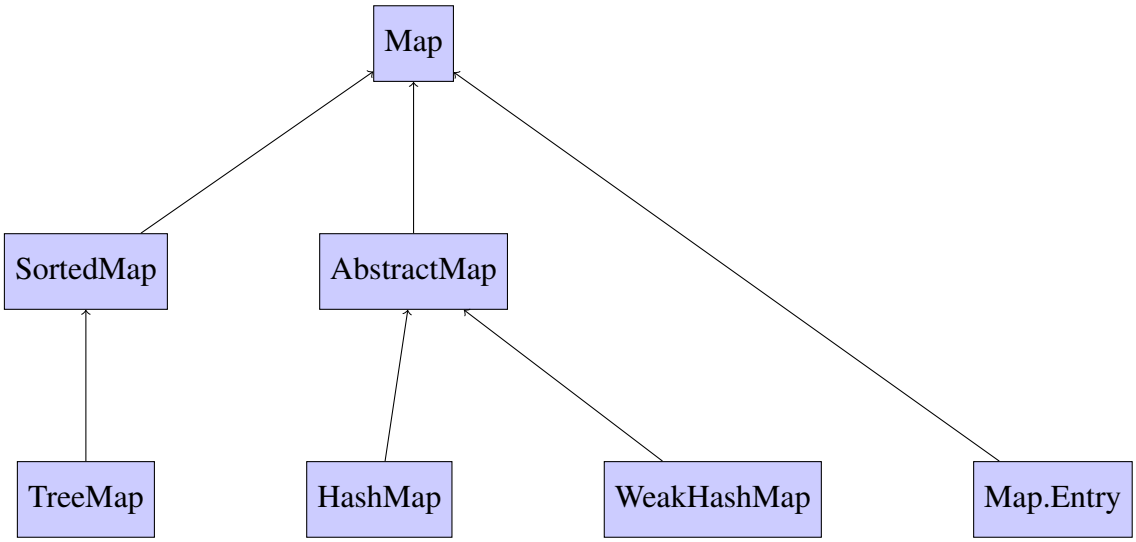
In Java, you can load these settings using the `Properties` class:

```
Properties props = new Properties();
FileInputStream in = new FileInputStream("settings.properties");
props.load(in);
String username = props.getProperty("username");
String password = props.getProperty("password");
```

## 8.4 Map

The `Map` interface is a part of the Java Collections framework and is introduced as a replacement for the `Dictionary` class. While the `Dictionary` class is abstract, it was paramount for its methods to be encapsulated within an interface, which is achieved by the `Map` interface. The `Map` interface facilitates operations with key-value pairs.

Interestingly, even though it's within the Collections framework, `Map` does not extend the `Collection` interface. Instead, it stands as the root of a separate hierarchy. Four primary classes implement the `Map` interface: `HashMap`, `WeakHashMap`, `TreeMap`, and `Hashtable`. Every element contained within these collections is of the type `Map.Entry`.




The above figure elucidates the hierarchy of these various classes and interfaces within the Java Collections framework.

### 8.4.1 Interface Map.Entry

The elements of a 'Map' are of type 'Map.Entry', an interface contained within 'Map'. Every key-value pair is an instance of this interface. We won't directly create instances of this, but the concrete classes will return objects that have already implemented this interface.

Below are the functions of this interface:

| Method | Description |
|---|---|
| `equals()` | checks equality with another object |
| `getKey()` | returns the key from the 'Map.Entry' |
| `getValue()` | returns the value from the 'Map.Entry' |
| `hashCode()` | computes the hash code for the current object |
| `setValue()` | changes the value in the 'Map.Entry' |

Consider a ‘HashMap‘ that maps names to their respective ages. The code below initializes such a map and then iterates over its entries, printing each key and its associated value:

```
import java.util.HashMap;
import java.util.Map;

public class MapEntryExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " +
    entry.getValue());
        }
    }
}
```

Here, ‘entrySet()‘ provides a set view of the mappings in the map, with each mapping being an instance of the ‘Map.Entry‘ interface.

## TreeMap

The `TreeMap` class is one of the implementations of the `Map` interface. It stores key-value pairs in a red-black tree, which is a self-balancing binary search tree. This ensures that the keys are always in sorted order. Apart from the common methods provided by the `Map` interface, `TreeMap` has additional methods that are useful due to its nature.

1. **firstKey()**: This method returns the first (smallest) key currently in the map.
2. **headMap(K toKey)**: It gives a view of the portion of the map whose keys are strictly less than `toKey`.
3. **lastKey()**: Returns the last (greatest) key currently in the map.
4. **subMap(K fromKey, K toKey)**: Provides a view of the portion of the map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.
5. **tailMap(K fromKey)**: Returns a view of the portion of the map whose keys are greater than or equal to `fromKey`.

Here, the TreeMap organizes the integers in ascending order, and then we use the special methods to obtain various subviews of the map.

```
import java.util.TreeMap;

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

```
5           TreeMap<Integer, String> treeMap = new TreeMap<>();
6           treeMap.put(5, "Five");
7           treeMap.put(1, "One");
8           treeMap.put(3, "Three");
9           treeMap.put(4, "Four");
10
11          System.out.println("First Key: " + treeMap.firstKey());    // Outputs
    : 1
12          System.out.println("Last Key: " + treeMap.lastKey());      // Outputs
    : 5
13          System.out.println("Head Map: " + treeMap.headMap(4));     // Outputs
    : {1=One, 3=Three}
14          System.out.println("Tail Map: " + treeMap.tailMap(3));     // Outputs
    : {3=Three, 4=Four, 5=Five}
15          System.out.println("Sub Map: " + treeMap.subMap(2, 5));    // Outputs
    : {3=Three, 4=Four}
16      }
17  }
```

## 8.5 Collections Class

The `Collections` class in Java's framework consists of static methods and objects designed to facilitate work with collections. The counterpart that operates on arrays is the `Arrays` class.

A description of the members of the `Collections` class is provided below.

1. **EMPTY_LIST**: Represents an immutable empty list.
2. **EMPTY_MAP**: Represents an immutable empty map.
3. **EMPTY_SET**: Represents an immutable empty set.
4. **binary_search()**: Searches for an element in a list using binary search technique.
5. **copy()**: Copies elements between two lists.
6. **enumeration()**: Converts a list to an Enumeration.
7. **fill()**: Fills the list with a particular element.
8. **max()**: Finds the maximum element in a collection.
9. **min()**: Finds the minimum element in a collection.
10. **nCopies()**: Creates an immutable list with multiple copies of a particular element.
11. **reverse()**: Reverses the elements in a list.
12. **reverseOrder()**: Returns a comparator that reverses the order of elements.

13. **shuffle()**: Randomly reorders the elements.
14. **singleton()**: Returns an immutable set with one element.
15. **singletonList()**: Returns an immutable list with one element.
16. **singletonMap()**: Returns an immutable map with one key-value pair.
17. **sort()**: Reorders elements in a list.

Apart from these, there are also methods that create thread-safe collections, but they won't be discussed here. Next, a few of the most important methods will be discussed in more detail.

## Sorting

The `sort()` method allows for the sorting of elements in a list:

```
public static void sort(List list);
public static void sort(List list, Comparator comp);
```

Here's an example where we will:

1. Create a list of integers.
2. Sort the list.
3. Reverse the list.
4. Search for an element using binary search.
5. Find the maximum and minimum of the list.
6. Shuffle the list randomly.

```
import java.util.*;

public class CollectionsExample {
    public static void main(String[] args) {
        // 1. Create a list of integers
        List<Integer> numbers = new ArrayList<>(Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5));

        // 2. Sort the list
        Collections.sort(numbers);
        System.out.println("Sorted list: " + numbers);

        // 3. Reverse the list
        Collections.reverse(numbers);
        System.out.println("Reversed list: " + numbers);

        // 4. Search for an element (5) using binary search
```

```
        // Important: binary_search requires the list to be sorted first!
        Collections.sort(numbers);
        int index = Collections.binarySearch(numbers, 5);
        System.out.println("Index of 5: " + index);

        // 5. Find the maximum and minimum of the list
        int max = Collections.max(numbers);
        int min = Collections.min(numbers);
        System.out.println("Max: " + max + ", Min: " + min);

        // 6. Shuffle the list randomly
        Collections.shuffle(numbers);
        System.out.println("Shuffled list: " + numbers);
    }
}
```

Expected output:

```
Sorted list: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
Reversed list: [9, 6, 5, 5, 5, 4, 3, 3, 2, 1, 1]
Index of 5: 7
Max: 9, Min: 1
Shuffled list: [some random permutation of numbers]
```

In this chapter, we've delved deeper into the advanced aspects of Java's Collections Framework. We covered advanced topics like concurrency considerations, custom comparators, the Collections utility class, and advanced collection types like `HashTable`, `TreeMap`, and `TreeSet`.

Understanding these advanced features not only broadens your toolkit but also equips you with the necessary skills to tackle complex data management problems in a Java application. These advanced structures and methods allow you to write code that is more efficient, flexible, and suitable for concurrent operations Weiss (2012).

# Exercises

## Exercise 1: Basic Map Usage

Write a Java program that uses a `HashMap` to store the names of students as keys and their scores as values. Implement the following operations:

- Add several student names and their scores to the map.
- Print the scores of all the students.
- Find and print the highest-scoring student.

## Exercise 2: TreeMap Operations

Write a Java program that uses a `TreeMap` to store a collection of books, where the book title is the key and the author is the value. Implement the following operations:

- Add several books to the map.
- Print the books in alphabetical order of their titles.
- Find and print the author of the book with the earliest title alphabetically.

### 8.5.1 Exercise 3: Using the `Collections` Class

Write a Java program that demonstrates the use of methods from the `Collections` class. Implement the following operations:

- Create a list of integers and add some elements to it.
- Use `Collections.sort()` to sort the list in ascending order.
- Use `Collections.reverse()` to reverse the order of the list.
- Use `Collections.shuffle()` to shuffle the elements randomly.
- Find and print the maximum and minimum values in the list using `Collections.max()` and `Collections.min()`.

### 8.5.2 Exercise 4: Counting Word Occurrences

Write a Java program that reads a text file, counts the occurrences of each word, and stores the word counts in a `HashMap`. Implement the following:

- Read a text file and process its contents to count word occurrences.
- Print the word and its count in alphabetical order of the words.
- Find and print the word with the highest count.

### 8.5.3 Exercise 5: Using Properties

Write a Java program that loads application settings from a properties file using the `Properties` class. Implement the following:

- Create a properties file (e.g., `settings.properties`) with key-value pairs.
- Load the properties from the file and retrieve specific settings.
- Print the retrieved settings (e.g., username and password).

### 8.5.4 Exercise 6: Map Entry

Write a Java program that demonstrates the use of the `Map.Entry` interface. Implement the following:

- Create a `HashMap` that maps country names to their populations (as integers).
- Iterate through the map entries and print both the country name and population for each entry.

# Chapter 9

# Java Application Development

This chapter covers the development of Java applications, including graphical user interfaces (GUIs), web applications, and mobile applications.

## 9.1 Graphical User Interfaces

Graphical User Interfaces (GUIs) provide an intuitive and user-friendly way for users to interact with Java applications. JavaFX is a popular library for creating GUIs in Java applications Vos et al. (2017). In this section, we will explore some of the key concepts and tools used in Java GUI development.

### 9.1.1 JavaFX Basics

JavaFX applications are built around the concept of stages and scenes. A stage represents a top-level container for your application's window, while a scene represents the content within that window.

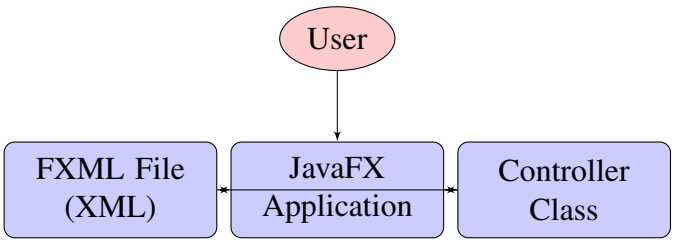



Figure 9.1: JavaFX Overview

In Figure 9.1.1, the FXML working principle is depicted. One may find the controller (code behind) as well as the defined GUI components (FXML) and how the user interacts with these via the FX application.

Here's a simple JavaFX application:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class HelloWorldApp extends Application {
    @Override
    public void start(Stage primaryStage) {
        // Create a button with "Hello, JavaFX!" text
        Button btn = new Button("Hello, JavaFX!");

        // Create a scene with the button as its root
        Scene scene = new Scene(btn, 300, 200);

        // Set the scene for the primary stage
        primaryStage.setScene(scene);

        // Set the title of the window
        primaryStage.setTitle("JavaFX Hello World");

        // Show the window
        primaryStage.show();
    }

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

### 9.1.2 JavaFX Layouts

JavaFX provides a set of layout containers that can be used to arrange GUI components in a specific way. Some of the commonly used layouts include:

- **BorderPane**: Divides the screen into five regions (top, bottom, left, right, and center) and allows components to be placed in each region.
- **HBox**: Arranges components in a horizontal row.
- **VBox**: Arranges components in a vertical column.
- **GridPane**: Arranges components in a grid with rows and columns.
- **StackPane**: Allows components to be stacked on top of each other.

Here is an example of using the VBox layout to arrange three components vertically:

```
VBox vbox = new VBox();
Label label1 = new Label("Label 1");
Label label2 = new Label("Label 2");
```

```
Button button = new Button("Click me");
vbox.getChildren().addAll(label1, label2, button);
```

This creates a VBox layout container and adds three components to it: two labels and a button. The components are arranged vertically in the order they were added.

### 9.1.3 Event Handling

JavaFX provides a powerful event handling mechanism that allows GUI components to respond to user actions, such as clicking a button or typing in a text field. Here is an example of adding an event handler to a button:

```
Button button = new Button("Click me");
button.setOnAction(e -> {
System.out.println("Button clicked");
});
```

This adds an event handler to the button that prints a message to the console when the button is clicked.

### 9.1.4 FXML

FXML is an XML-based markup language used to define JavaFX GUI layouts. It allows GUI components to be defined in a separate file from the Java code, making it easier to maintain and modify the GUI. Here is an example of an FXML file that defines a simple GUI:

```

<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<VBox xmlns="http://javafx.com/javafx"
   xmlns:fx="http://javafx.com/fxml"
   fx:controller="com.example.MyController">
<Label text="Hello, World!"/>
<Button text="Click me" onAction="#handleButtonClick"/>
</VBox>
```

This defines a VBox layout container with a label and a button. The onAction attribute of the button specifies the name of the method to be called when the button is clicked. The corresponding Java code for this FXML file would look like this:

```
public class MyController {
@FXML private Label label;
@FXML private Button button;
private void handleButtonClick(ActionEvent event) {
    label.setText("Button clicked");
}
}
```

This defines a Java class with an event handler method that sets the text of the label to ”Button clicked” when the button is clicked.

## 9.2 Web Applications

This section covers the development of web applications in Java, using tools such as the Spring Framework and JavaServer Pages (JSP) Walls (2016).

- Introduction to web applications
- Overview of web frameworks
- Setting up a web application development environment
- Servlets and JSPs
- Introduction to Spring Framework
- Building web applications with Spring MVC
- Building RESTful web services with Spring
- Introduction to Hibernate ORM
- Persistence with Hibernate and Spring
- Building a web application with Spring, Hibernate, and MySQL
- Deploying web applications to a server

The section would provide an introduction to web application development and discuss the most common web frameworks used in Java development. It would cover how to set up a development environment and provide examples using popular web frameworks like Spring and Hibernate. The section would also discuss building RESTful web services and deploying applications to a server.

### 9.2.1 Servlets and JSPs

Servlets and JSPs are the building blocks of Java web applications. Servlets handle processing, while JSPs handle presentation.

For instance, consider a simple servlet for handling form submissions:

```
@WebServlet("/formHandler")
public class MyServlet extends HttpServlet {
protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
String name = request.getParameter("name");
response.getWriter().write("Hello, " + name);
}
}
```

And a corresponding form in a JSP:

```
<form action="formHandler" method="post">
    Name: <input type="text" name="name" />
    <input type="submit" value="Submit" />
</form>
```

### 9.2.2 Introduction to Spring Framework

The Spring Framework simplifies Java development, offering tools and libraries for creating scalable and secure applications. Spring MVC, a subset of the framework, is ideal for web application development.

For instance, a basic Spring controller might look like this:

```
@RestController
@RequestMapping("/greeting")
public class GreetingController {
@GetMapping
public String greet(@RequestParam(value="name", defaultValue="World") String
    name) {
return "Hello, " + name;
}
}
```

## Complex Spring Boot Sample: Book Management System

Let's see how to develop a more complex Spring Boot application that integrates Spring MVC, Spring Data JPA, and an H2 database. This sample will create a RESTful API for a basic book management system.

#### Core Features of Maven

- **Convention Over Configuration**: Maven has a standard project structure. If you follow this structure, you require minimal configurations for your project.
- **Dependencies Management**: Using the POM (Project Object Model) file, Maven manages project dependencies and can automatically download the required libraries from the central Maven repository.
- **Lifecycle**: Maven follows a lifecycle approach. This means that the process of cleaning, compiling, and packaging the code is standardized.
- **Plugins**: Maven has a rich set of plugins which can be used to run unit tests, generate reports, and more.
- **Repositories**: Maven repositories store jar files, libraries, plugins, and other build artifacts that can be shared among projects.

### Maven's POM File

The heart of a Maven project resides in the POM file (usually named `pom.xml`). This XML file contains information about the project and configuration details utilized by Maven to build the project. It includes:

- Project dependencies.
- Plugins.
- Goals and phases.
- Project version.
- Project description, URL, and other metadata.

To include a dependency in a Maven project, you add it to the POM. For example, to include the JUnit library, you would add the following:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>
    <scope>test</scope>
</dependency>
```

With this configuration, Maven would automatically fetch the specified version of JUnit from the central repository and make it available in the project.

### Building with Maven

Maven uses commands (often termed as "goals") to execute specific lifecycle phases. Some commonly used Maven commands include:

- `mvn clean`: Removes the `target/` directory created previously during the build, ensuring a clean slate.
- `mvn compile`: Compiles the source code.
- `mvn test`: Runs the tests.
- `mvn package`: Packages the compiled code into its distributable format (e.g., JAR).
- `mvn install`: Installs the package into the local repository.

### Maven Dependencies

The necessary dependencies in our `pom.xml` are:

```
<dependencies>
    <!-- Spring Boot Web Starter for REST API -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Boot Data JPA Starter for database operations -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- H2 Database for in-memory database -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

### Entity Class

Define a `Book` entity:

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String author;

    // Getters, setters, and other boilerplate methods...
}
```

### Repository Interface

Create a JPA repository:

```
public interface BookRepository extends JpaRepository<Book, Long> {
}
```

### Service Class

Define the service layer:

```
@Service
public class BookService {

    @Autowired
    private BookRepository bookRepository;

    public List<Book> getAllBooks() {
        return bookRepository.findAll();
    }

    public Book getBookById(Long id) {
        return bookRepository.findById(id).orElse(null);
    }

    public Book saveBook(Book book) {
        return bookRepository.save(book);
    }

    public void deleteBook(Long id) {
        bookRepository.deleteById(id);
    }
}
```

### REST Controller

Implement a controller to manage books:

```
@RestController
@RequestMapping("/api/books")
public class BookController {

    @Autowired
    private BookService bookService;

    @GetMapping
    public List<Book> getAllBooks() {
        return bookService.getAllBooks();
    }

    @GetMapping("/{id}")
    public Book getBookById(@PathVariable Long id) {
        return bookService.getBookById(id);
    }

    @PostMapping
    public Book createBook(@RequestBody Book book) {
        return bookService.saveBook(book);
    }
```

```
22
23      @PutMapping("/{id}")
24      public Book updateBook(@PathVariable Long id, @RequestBody Book book) {
25          Book existingBook = bookService.getBookById(id);
26          BeanUtils.copyProperties(book, existingBook, "id");
27          return bookService.saveBook(existingBook);
28      }
29
30      @DeleteMapping("/{id}")
31      public void deleteBook(@PathVariable Long id) {
32          bookService.deleteBook(id);
33      }
34  }
```

### Application Properties

Configure `application.properties`:

```
spring.datasource.url=jdbc:h2:mem:bookdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
```

### Spring Boot Application Runner

Create the main Spring Boot application:

```
@SpringBootApplication
public class BookApplication {

    public static void main(String[] args) {
        SpringApplication.run(BookApplication.class, args);
    }
}
```

Now start the application and you will have a RESTful API for a book management system. You can use tools like Postman to test the CRUD operations, or you can run the application directly in Intellij in debug mode to control the execution more thoroughly.

## 9.3 Exercises

### 9.3.1 JavaFX

**Exercise 1: Creating a JavaFX GUI**

**Objective:** Build a simple JavaFX GUI application.

1. Create a new JavaFX project in your preferred IDE or build tool.
2. Design a JavaFX GUI with a window containing a label and a button.
3. Write the Java code to display ”Hello, JavaFX!” when the button is clicked.
4. Test your JavaFX application.

**Exercise 2: Working with JavaFX Layouts**

**Objective:** Practice using JavaFX layout containers.

1. Create a JavaFX application with a BorderPane layout.
2. Add various UI elements (e.g., labels, buttons, text fields) to different regions of the BorderPane (top, bottom, left, right, center).
3. Implement an event handler to respond to user interactions.
4. Test and run your JavaFX layout application.

**Exercise 3: Building a To-Do List Application**

**Objective:** Develop a JavaFX application for managing a to-do list.

1. Create a JavaFX project.
2. Design a user interface that allows users to add, edit, and remove tasks.
3. Implement the backend logic to store and manage tasks.
4. Connect the user interface with the backend logic to create a functional to-do list app.

### 9.3.2 Web Applications

**Exercise 4: Setting Up a Spring Boot Project**

**Objective:** Set up a basic Spring Boot project for web development.

1. Create a new Spring Boot project using your preferred IDE or Spring Initializer (`https://start.spring.io/`).
2. Configure the project to use Spring Web and Thymeleaf (a template engine).
3. Create a simple Spring MVC controller that handles requests to ”/hello” and displays a ”Hello, World!” message using a Thymeleaf template.
4. Run the Spring Boot application and access the ”/hello” URL in your web browser.

**Exercise 5: Building a RESTful API**

**Objective:** Create a RESTful API using Spring Boot.

1. Expand the previous Spring Boot project.
2. Create a new entity class, e.g., ”Product,” with attributes like id, name, description, and price.
3. Implement a repository interface for the ”Product” entity.
4. Create a RESTful controller that allows users to perform CRUD operations (Create, Read, Update, Delete) on the ”Product” entity.
5. Test your API using tools like Postman or cURL to create, retrieve, update, and delete products.

**Exercise 6: Working with Databases**

**Objective:** Integrate Spring Data JPA with your Spring Boot project.

1. Extend the ”Product” entity to include JPA annotations for mapping it to a database table.
2. Configure a database connection (e.g., H2 or MySQL) in your `application.properties` or `application.yml` file.
3. Use Spring Data JPA to persist and retrieve ”Product” entities from the database.
4. Implement additional API endpoints to demonstrate database operations, such as searching for products by name or price range.

**Exercise 7: Building a Secure Web Application**

**Objective:** Add security to your web application using Spring Security.

1. Include Spring Security as a dependency in your project.
2. Configure Spring Security to require authentication for certain URL patterns.
3. Create user accounts and roles.
4. Implement user registration and login functionality.
5. Restrict access to specific parts of your application based on user roles.

**Exercise 8: Implementing a File Upload Feature**

**Objective:** Allow users to upload files to your web application.

1. Extend your Spring Boot project to handle file uploads.
2. Create an endpoint that accepts file uploads from users.
3. Implement file upload validation, including file type and size checks.
4. Save the uploaded files to a designated directory or a database.

**Exercise 9: Building a RESTful Web Service**

**Objective:** Create a RESTful web service using Spring Boot.

1. Create a new Spring Boot project for building a RESTful web service.
2. Define a resource (e.g., ”Employee”) with attributes such as id, name, and department.
3. Implement CRUD operations for managing employees via RESTful endpoints (GET, POST, PUT, DELETE).
4. Test the web service using API testing tools like Postman or by writing a simple client application.

# Chapter 10

# Java Tools and Libraries

This chapter covers the tools and libraries commonly used in Java development, including Integrated Development Environments (IDEs), build tools, and third-party libraries.

## 10.1 Tools

### 10.1.1 Integrated Development Environments

This section covers popular IDEs for Java development, such as Eclipse Burnette (2005), NetBeans Wielenga (2015), and IntelliJ IDEA Fields et al. (2006).

### 10.1.2 Build Tools

This section covers build tools for Java, such as Maven and Gradle, which automate the process of building and deploying Java applications.

### 10.1.3 Third-Party Libraries

- Testing frameworks: JUnit and Mockito
- Logging frameworks: Log4j and SLF4J
- Application servers: Tomcat and Jetty
- Database access: JDBC and Hibernate
- Messaging frameworks: JMS and RabbitMQ
- JSON processing: Jackson and Gson
- XML processing: DOM and SAX
- Web scraping: Jsoup and Selenium

- Machine learning libraries: Weka and TensorFlow

## 10.2 Network Communication in Java

Java offers robust libraries for network communication, allowing developers to create networked applications with relative ease Harold (2000). The primary package for these operations is `java.net`.

### 10.2.1 Sockets

At the heart of network communication in Java are sockets. A socket is one endpoint of a two-way communication link between two programs running on the network.

#### TCP Sockets

TCP is a connection-oriented protocol, meaning there's a persistent connection between the client and the server.

- **ServerSocket**: This class is used for creating servers that listen for incoming connections.
- **Socket**: Represents the client side of a connection, but it's also used on the server side to represent an individual connection to a client.

**Example - TCP Server**

```
import java.io.*;
import java.net.*;

public class TCPServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(8888);
        Socket clientSocket = serverSocket.accept();

        PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),
    true);
        BufferedReader in = new BufferedReader(new InputStreamReader(
    clientSocket.getInputStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            out.println(inputLine);
        }
    }
}
```

**Example - TCP Client**

```java
import java.io.*;
import java.net.*;

public class TCPClient {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("localhost", 8888);

        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

        out.println("Hello Server");
        System.out.println("Server replied: " + in.readLine());
    }
}
```

### UDP Sockets

Unlike TCP, UDP is connectionless. Java provides the ‘DatagramSocket‘ class for UDP communication.

#### Example - UDP Server

```java
import java.net.*;

public class UDPServer {
    public static void main(String[] args) throws IOException {
        DatagramSocket serverSocket = new DatagramSocket(8888);
        byte[] receiveData = new byte[1024];

        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        serverSocket.receive(receivePacket);

        String message = new String(receivePacket.getData(), 0, receivePacket.getLength());
        System.out.println("Received: " + message);
    }
}
```

#### Example - UDP Client

```java
import java.net.*;

public class UDPClient {
    public static void main(String[] args) throws IOException {
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress serverAddress = InetAddress.getByName("localhost");

        String message = "Hello Server";
        byte[] sendData = message.getBytes();

```

```
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.
    length, serverAddress, 8888);
        clientSocket.send(sendPacket);
    }
}
```

### 10.2.2 URL and HttpURLConnection

For web-related tasks, Java offers classes like ‘URL‘ and ‘HttpURLConnection‘ to read content from URLs.

**Example - Reading Content from a URL**

```
import java.io.*;
import java.net.*;

public class URLReader {
    public static void main(String[] args) throws IOException {
        URL url = new URL("http://www.example.com/");
        BufferedReader in = new BufferedReader(new InputStreamReader(url.
    openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
        }
        in.close();
    }
}
```

In summary, Java’s networking capabilities are extensive, allowing for the development of various networked applications, from simple socket-based utilities to complex web service clients and servers.

## 10.3 Database Access in Java

Java offers powerful tools and libraries for interacting with databases. The two prominent methods are JDBC and Hibernate ORM.

### 10.3.1 Java Database Connectivity (JDBC)

JDBC is a Java-based API that allows Java applications to interact with databases. It provides methods for querying and updating data in a database.

#### JDBC Workflow

1. Load the database driver.

2. Establish a connection to the database.
3. Create a statement.
4. Execute the statement and retrieve results.
5. Process the results.
6. Close the connection.

**Example - Connecting to a Database and Retrieving Data**

```
import java.sql.*;

public class JDBCExample {
    public static void main(String[] args) {
        try {
            // Load the driver
            Class.forName("com.mysql.jdbc.Driver");

            // Establish connection
            Connection conn = DriverManager.getConnection("jdbc:mysql://
    localhost:3306/mydatabase", "user", "password");

            // Create statement
            Statement stmt = conn.createStatement();

            // Execute query
            ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");

            // Process results
            while(rs.next()) {
                System.out.println(rs.getInt("id") + "  " + rs.getString("
    name"));
            }

            // Close connection
            conn.close();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

### 10.3.2 Hibernate ORM

Hibernate is an Object-Relational Mapping (ORM) framework. It maps Java objects to database tables and vice versa. Hibernate abstracts the database operations, letting developers work with objects rather than SQL statements.

### Hibernate Workflow

1. Define Entity classes (Java classes that represent tables in the database).
2. Create Hibernate configuration file and mapping files (or use annotations).
3. Obtain a ‘SessionFactory‘.
4. Start transactions using a ‘Session‘ object.
5. Perform CRUD operations.
6. Commit transactions and close the session.

**Example - Simple Hibernate Usage**
Given an entity class ‘Person‘:

```
@Entity
@Table(name = "person")
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

    @Column(name = "name")
    private String name;

    // Getters, setters, and constructors
}
```

The following code demonstrates how to save a ‘Person‘ object to the database:

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

Person person = new Person();
person.setName("John");
session.save(person);

transaction.commit();
session.close();
```

In summary, while JDBC provides a low-level API for database interactions, Hibernate offers a higher-level approach, allowing developers to work more directly with Java objects, abstracting most of the database-related code.

## 10.4 Messaging Frameworks in Java

Distributed systems often require components to communicate asynchronously. Java provides various tools and libraries for messaging. Prominent among these are JMS and RabbitMQ.

### 10.4.1 Java Message Service (JMS)

JMS is a Java API for sending and receiving messages between two or more clients. It allows for the creation of message-oriented middleware applications using the JavaEE platform.

#### JMS Components

- **Message Producers**: Components responsible for sending messages.
- **Message Consumers**: Components that receive the messages.
- **Message Queue**: A staging area that contains messages sent by the producer application and waits to be read by a consumer application.
- **Message Topic**: Enables the publish/subscribe messaging model.

**Example - Sending a Message using JMS**

```
Context context = new InitialContext();
Queue queue = (Queue) context.lookup("java:comp/env/jms/queue");
ConnectionFactory connectionFactory = (ConnectionFactory) context.lookup("
    java:comp/env/jms/connectionFactory");
Connection connection = connectionFactory.createConnection();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageProducer producer = session.createProducer(queue);
TextMessage message = session.createTextMessage("Hello, World!");
producer.send(message);
session.close();
connection.close();
```

### 10.4.2 RabbitMQ

RabbitMQ is an open-source message-broker software that acts as an intermediary for sending and receiving messages. It uses the Advanced Message Queuing Protocol (AMQP) but also supports other protocols.

#### RabbitMQ Components

- **Exchanges**: Receive messages from publishers and push them to queues depending on rules (bindings) defined.
- **Queues**: Buffers that store messages.

- **Bindings**: Rules that exchanges use to route messages to queues.
- **Producers**: Send messages to an exchange.
- **Consumers**: Receive messages from a queue.

**Example - Sending a Message using RabbitMQ with Spring Boot**
First, include the Spring Boot RabbitMQ starter in your ‘pom.xml‘:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Then, in Java:

```
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class MessageSenderService {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void sendMsg(String message) {
        rabbitTemplate.convertAndSend("exchangeName", "routingKey", message)
    ;
     }
}
```

Both JMS and RabbitMQ provide mechanisms for asynchronous communication. While JMS is Java-centric, RabbitMQ offers a more language-agnostic approach, making it suitable for cross-language messaging.

### 10.4.3 Other Messaging Frameworks

There are more libraries that provide even more sophisticated messaging mechanism and the most popular ones are described below.

#### Apache Kafka

Apache Kafka is a distributed event streaming platform originally built by LinkedIn and donated to the Apache Software Foundation. It’s designed for high-throughput and fault-tolerant real-time event streaming and is used by many large-scale systems.

- **Producers**: Send records to topics.
- **Consumers**: Read records from topics.

- **Topics**: Channels where records are sent by producers.
- **Partitions**: Each topic can be split into partitions to enhance parallelism.

**Use Case**: Kafka is popularly used for real-time analytics, monitoring, log aggregation, and more.

### Apache ActiveMQ

Apache ActiveMQ is an open-source message broker that supports JMS as well as other messaging protocols. It's known for its performance and flexibility.

- **Topics and Queues**: Supports both pub/sub and point-to-point messaging models.
- **Brokers**: Handles the receipt, storage, and dispatching of messages.

**Use Case**: Often used in enterprise systems to integrate different applications and allow them to communicate asynchronously.

### ZeroMQ

ZeroMQ (also known as ØMQ) is a high-performance messaging library, designed to be used in distributed or concurrent applications. It's different from traditional messaging brokers, as it's brokerless; the logic is handled in the clients.

- **Sockets**: ZeroMQ sockets provide a mechanism for asynchronous message passing.

**Use Case**: Used in scenarios where low latency and performance are critical and the system can operate without a centralized broker.

### NATS

NATS is a lightweight and high-performance messaging system. It's designed for modern cloud-native and IoT applications.

- **Publishers and Subscribers**: NATS follows a pub/sub model.
- **Subjects**: The channels or topics to which publishers send or from which subscribers receive messages.

**Use Case**: Commonly used in microservices architectures due to its simplicity and performance.

### Pulsar

Apache Pulsar is a cloud-native messaging and event streaming platform that provides multi-tenancy, persistent storage, and real-time event streaming.

- **Producers and Consumers**: Follow a pub/sub model.
- **Topics and Subscriptions**: Allow for message retention and replay.

**Use Case**: Suitable for real-time event streaming, data pipelines, and integrating different components of a distributed system.

In conclusion, the choice of messaging system often depends on the specific needs of the application, such as the volume of messages, the need for persistence, latency requirements, and more. Each of the above frameworks and brokers has its strengths and is designed to address particular challenges in distributed systems.

# Chapter 11

# Laboratory Exercises

The laboratories were crafted with the intention of providing readers with practical exposure to Java programming, facilitating a smoother transition into high-level programming languages.

## 11.1 Laboratory 1: Introduction

This laboratory serves as an introduction to Java programming. Before diving into the exercises, you should set up your development environment. Please first read Chapter 2: Overview of Java.

### 11.1.1 Development Environments

- Eclipse: An open-source Integrated Development Environment used primarily for Java. Eclipse comes with a variety of plugins for developing web applications, C/C++ applications, and much more.

- NetBeans: Another open-source IDE, often used for Java, JavaScript, PHP, and other languages. It is known for its simple and straightforward interface.

- IntelliJ IDEA: A powerful IDE developed by JetBrains, IntelliJ IDEA offers advanced coding, debugging, and refactoring features. It has a community version that's free and a professional version with more features but requires a subscription.

- Oracle JDeveloper: A free IDE provided by Oracle, primarily aimed at Java, Java EE and Oracle ADF development.

- Visual Studio Code: A free, open-source editor developed by Microsoft. While not a full-fledged IDE, it is highly extensible and supports Java development via plugins.

- BlueJ: Aimed primarily at beginners, BlueJ offers a simpler interface and is specifically designed for teaching and learning Java.

### 11.1.2 A Simple Example

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

**Requirements**

1. Compile the `Example.java` file using the `javac` utility.
2. Run the example using the `java` command.
3. Modify the program to display the command line arguments using the `args` parameter.

### 11.1.3 Variables

In order to understand the contents of this section it is mandatory to read Chapter 3: Basic Java Programming: Syntax and structures.

```
class Vars {
    public static void main(String args[]) {
        int x = 150;
        double y = 2.0;
        float z = 20f;
        byte b = 10;
        short c = 14;
        y = x;
        System.out.println(y);
    }
}
```

**Requirements**

1. Compile and run the code.
2. Experiment with converting one variable type to another, similar to the instruction `y = x`. Note your observations.
3. Test the same using wrapper classes like `Double` instead of `double`. Use methods like `intValue()` or `byteValue()` for type conversion.
4. Try type casting using parsing functions like `valueOf()` or `parseDouble()`.
5. Test other methods of wrapper classes, such as `isNaN()` or `compare()`.

### 11.1.4 Arrays

```
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
String months[] = { "January", "February", "March", "April", "May", "June",
                    "July", "August", "September", "October", "November", "
   December" };
```

**Exercises**

1. Display each month alongside its corresponding number of days.
2. Display the months that have 31 days.
3. Display the months that have fewer than 31 days, along with their number of days.
4. Calculate and display the total number of days for months with fewer than 31 days.
5. Repeat exercise 1, but only display the first 3 characters of each month's name (e.g., Jan, Feb, etc.).

## 11.2 Laboratory 2: Object Oriented Programming

### 11.2.1 Introduction

This lab focuses on object-oriented programming in Java. We will work with classes such as `Curs` (Course), `Profesor` (Professor), `Student`, and `ManagerCursuri` (Course Manager). We will also deal with database operations. In order to understand the tasks from this laboratory, it is required to read Chapter 4: Object-Oriented Programming in Java.

### 11.2.2 Project Description

We are provided with several classes: `Curs`, `Profesor`, `Student`, and `ManagerCursuri`. The aim is to add functionalities to:

- Add/Remove/Modify a course in the array `Curs[] cursuri` within `ManagerCursuri`.
- Remove/Modify a student within `Course`.
- Add/Remove/Modify a professor within `Course`.
- Report all students enrolled in a course.
- Report all courses.
- Allow a professor to grade a student.
- Report the grades of all students.

- Report the average grades of students in a course.
- Report the average grades given by a professor.

### 11.2.3 Code Snippets

#### Course Class (`Curs`)

```java
package classes;
public class Course {
    String name;
    String description;
    Professor teacher;
    Student[] students;

    public Course(String name, String description,
    Professor teacher, Student[] students) {
        this.name = name;
        this.description = description;
        this.teacher = teacher;
        this.students = students;
    }

    public void updateProfessor(Professor teacher) {
        this.teacher = teacher;
    }

    public void addStudent(Student student) {
        // Working with arrays, we need to insert by using an auxiliary
   array
        int newLength = students.length + 1;
        Student[] aux = new Student[newLength];
        int index = 0;
        for (Student s : students) {
            aux[index++] = s;
        }
        // Finally, we add the new student at the last index
        aux[index] = student;
        // And we reallocate the students list with aux;
        this.students = new Student[newLength];
        System.arraycopy(aux, 0, students, 0, newLength);
    }

    @Override
    public String toString() {
        String str = "Course: " + "name=" + name + ", description=" +
   description + ",\nTeacher=" + teacher + ",\nStudents:\n";
        for (Student s : students) {
            str += s + "\n";
        }
```

```
        return str;
    }
}
```

### Course Manager Class (`ManagerCursuri`)

```
package classes;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class CourseManager {
    Course[] courses;

    public CourseManager() {
        Connection conn = null;
        courses = new Course[0];
    }

    public void addCourse(Course course) {
        int newLength = courses.length + 1;
        Course[] aux = new Course[newLength];
        int index = 0;
        for (Course c : courses) {
            aux[index++] = c;
        }
        // Finally, we add the new course at the last index
        aux[index] = course;
        // And we reallocate the courses list with aux
        this.courses = new Course[newLength];
        System.arraycopy(aux, 0, courses, 0, newLength);
    }

    public void displayCoursesToConsole() {
        for (Course c : courses) {
            System.out.println(c);
        }
    }
}
```

### Professor Class (`Profesor`)

```
package classes;

public class Professor {
    String firstName;
```

```
    String lastName;

    @Override
    public String toString() {
        return "Professor{" + "firstName=" + firstName + ", lastName=" +
    lastName + '}';
    }

    public Professor(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

### Student Class (`Student`)

```
package classes;

public class Student {
    String firstName;
    String lastName;
    int groupNumber;

    public Student(String firstName, String lastName, int groupNumber) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.groupNumber = groupNumber;
    }

    @Override
    public String toString() {
        return "Student{" + "firstName=" + firstName + ", lastName=" +
    lastName + ", groupNumber=" + groupNumber + '}';
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
```

```
34
35      public int getGroupNumber() {
36          return groupNumber;
37      }
38
39      public void setGroupNumber(int groupNumber) {
40          this.groupNumber = groupNumber;
41      }
42  }
```

**Demo Classes (`Classes`)**

```
1  package classes;
2
3  public class Classes {
4      public static void main(String[] args) {
5          // Define students
6          Student[] students = new Student[]{new Student("Andrei", "Negoita",
       2231), new Student("Ion", "Mateescu", 4221)};
7
8          // Define professor
9          Professor prof = new Professor("Anton", "Panaitescu");
10
11         // Define new course
12         Course course = new Course("Material Resistance", "Calculations of
       reactions,\n" +
13                 "effort diagrams, calculations of geometric characteristics of\n
       " +
14                 "flat surfaces and calculations of resistance elements to simple
        stresses\n",
15                 prof, students);
16
17         // Add course to the list of courses
18         CourseManager courseManager = new CourseManager();
19         courseManager.addCourse(course);
20
21         // Display courses to the console
22         courseManager.DisplayCoursesToConsole();
23     }
24 }
```

### 11.2.4 Exercises

**Exercise 1: Add Methods to Student and Professor**

Add a method `getFullName()` to both the `Student` and `Professor` classes that returns the full name in the format “FirstName LastName”.

**Exercise 2: Dynamic Course Enrollment**

Add a method to `ManagerCursuri` (CourseManager) that allows you to enroll a student into a course by specifying the course name and the student object.

**Exercise 3: Course Reporting**

Extend `ManagerCursuri` with a method `listStudentsInCourse(String courseName)` that returns a list of all students enrolled in a given course.

**Exercise 4: Course Average**

Create a method in `CourseManager` that calculates and returns the average grade for a specific course.

**Exercise 5: Professor Grading Average**

Create a method in `CourseManager` that calculates and returns the average grade given by a specific professor across all their courses.

### 11.2.5 Additional Exercises: Database

Create a database (either H2, MSSQL or MySQL) named `Classes` with the following tables:

- Course
- Student
- Professor
- CoursesStudents (To specify that student X is participating in course Y)

Save the data (courses, students, etc.) from `ClassesDemo` into the database.

## 11.3 Laboratory 3: Inheritance

### 11.3.1 Introduction

Inheritance is one of the four fundamental OOP principles. The main idea behind inheritance is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields, and you can add new methods and fields to adapt your new class towards a specific need: Gamma et al. (1995). In order to perform well in this laboratory, it is necessary to read Chapter 5: Advanced Object-Oriented Programming in Java. In this laboratory exercise, we will explore the concept of inheritance in Java. We'll define a base class called `Person` and two derived classes named `Professor` and `Student`. These classes will demonstrate the basics of inheritance, method overriding, and constructor overloading.

### 11.3.2 Person Class

```
public abstract class Person {
    String name;
    String surname;

    @Override
    public String toString() {
        return "Person{" + "name=" + name + ", surname=" + surname + '}';
    }
}
```

### 11.3.3 Requirements

- Add the above class to the project you completed in the previous lab.
- Modify the Professor and Student classes so they extend the Person class. The Professor and Student classes will no longer need the 'name' and 'surname' members, and `toString()` will be overridden in each class.
- Overload the constructors in the Professor, Student, and Course classes, so that there's a default constructor in each class. The class members will be initialized with empty strings for names, zero for the group, or an empty list for students.

### 11.3.4 CourseOperations Interface

```
public interface CourseOperations {
    public void UpdateProfessor(Professor p);
    public void AddStudent(Student student);
    public void RemoveStudent(Student student);
    public void UpdateStudent(Student student);
    public void UpdateCourse(String name, String description);
}
```

### 11.3.5 Requirements for CourseOperations Interface

- Add the above interface to your project.
- Modify the Course class as follows: `public class Course implements CourseOperatio`
- Add functional code for the methods:
  - `public void RemoveStudent(Student student);`
  - `public void UpdateStudent(Student student);`
  - `public void UpdateCourse(String name, String description);`

### 11.3.6 ManagerCourseOperations Interface

```
public interface ManagerCourseOperations {
    public void AddCourse(Course course);
    public void UpdateCourse(Course course);
    public void DeleteCourse(Course course);
}
```

### 11.3.7 Requirements for ManagerCourseOperations Interface

- Add the above interface to your project.
- Modify the CourseManager class as follows: `public class CourseManager implements ManagerCourseOperations`.
- Add functional code for the methods:
  - `public void UpdateCourse(Course course);`
  - `public void DeleteCourse(Course course);`

### 11.3.8 Additional Exercises

- Modify the code to save, retrieve, and update data in a database, related to Course, Student, and Professor, following the above modifications.
- Solve the exercises at the end of Course 3.

## 11.4 Laboratory 4: Collections

### 11.4.1 Introduction

The power of object-oriented programming (OOP) lies not just in inheritance but also in the flexible management of collections of objects. While inheritance allows for an organized, hierarchical structure and code reuse, collections provide ways to manage groups of objects effectively. To fulfil the tasks from this laboratory, a prerequisite is to go through Chapter 6: Java Collections. In this lab exercise, we extend the functionality of a university management system introduced in previous labs. Specifically, we will move away from using arrays to more dynamic data structures like ‘Set‘ and ‘List‘, which are part of Java’s Collections Framework: Bloch (2018). The goal is to make the system more flexible and capable of handling various operations like sorting, grouping, and uniquely identifying students and courses.

### 11.4.2 Modify the Course Class

- Modify the ‘Course‘ class from previous labs such that the array ‘Student[] students‘ becomes a list of students, for example, ‘Set¡Student¿ students‘.
- Students should be uniquely identified by a composite key consisting of their name, surname, and group.

### 11.4.3 Modify the CourseManager Class

- Modify the ‘CourseManager‘ class such that the array ‘Course[] courses‘ becomes a list of courses: ‘List¡Course¿ courses‘.

### 11.4.4 Modify MockClassesManager to Make It Functional

Make necessary modifications to make ‘MockClassesManager‘ functional.

```
1  package classes;
2
3  import java.util.ArrayList;
4
5  public class MockClassesManager implements ManagerCourseOperations{
6
7      @Override
8      public void AddCourse(Course course){
9          throw new UnsupportedOperationException("Not supported yet."); //To
       change body of generated methods, choose Tools | Templates.
10     }
11
12     @Override
13     public void UpdateCourse(Course course){
14         throw new UnsupportedOperationException("Not supported yet."); //To
       change body of generated methods, choose Tools | Templates.
15     }
16
17     @Override
18     public void DeleteCourse(Course course){
19         throw new UnsupportedOperationException("Not supported yet."); //To
       change body of generated methods, choose Tools | Templates.
20     }
21
22     public ArrayList<Course> GenerateCourses() {
23         ArrayList<Course> courses = new ArrayList<>();
24
25         ArrayList<Student> students = new ArrayList<>();
26         students.add(new Student("Andrei","Negoita",2231));
27         students.add(new Student("Ion","Mateescu",4221));
28
29         Professor prof = new Professor("Anton","Panaitescu");
30
```

```
31          courses.add(new Course("Rezistenta Materialelor", "calculul
        reactiunilor,\n" + "diagramele de eforturi, calculul caracteristicilor
        geometrice ale\n" + "suprafetelor plane si calculul elementelor de
        rezistenta la solicitari simple\n" , prof, students));
32          return courses;
33      }
34
35  }
```

### 11.4.5 Additional Implementations

- Students should be easily grouped by their group number.
- Students should be displayable in order without implementing a sorting algorithm.
- Courses should be sortable by name, professor's name, or the number of enrolled students.

### 11.4.6 Example Modifications

```
1  // Defines an interface with a method to get a list of courses
2  public interface ISourceManager extends ManagerCourseOperations {
3      public List<Course> getCourses();
4  }
5
6  // Represents a course with a name, description, professor, and list of
       students
7  public class Course implements CourseOperations {
8      String name;
9      String description;
10     Professor professor;
11     Set<Student> students;
12     // ...
13 }
14
15 // Manages a list of courses
16 public class CourseManager implements ISourceManager {
17     List<Course> courses;
18
19     // ...
20 }
```

### 11.4.7 Exercises

#### Exercise 1: Implement a Grouping Mechanism

Implement a function in the 'CourseManager' class to group students by their group number.

**Exercise 2: Implement Natural Ordering for Students**

Override the ‘compareTo‘ method in the ‘Student‘ class to enable natural ordering based on students’ names.

**Exercise 3: Sort Courses**

Add a method in the ‘CourseManager‘ class to sort the list of courses by name, professor’s name, or the number of enrolled students. Use Java’s built-in sorting capabilities.

**Exercise 4: Add a Search Feature**

Implement a function in the ‘CourseManager‘ class that allows you to search for a course by its name.

**Exercise 5: Mock Data Population**

Use the ‘MockClassesManager‘ to populate your lists with mock data. Test all your implemented features to make sure they work as expected.

**Exercise 6: Database Integration (Optional)**

If you’re up for a challenge, integrate the database to save, retrieve, and modify data related to courses, students, and professors.

## 11.5 Laboratory 5: Exceptions and Generics

### 11.5.1 Introduction

Handling errors and reusability are fundamental aspects of robust software development. In this laboratory, we will focus on two essential concepts: Exceptions and Generics. Exception handling in Java is a powerful mechanism that is used for handling runtime errors, allowing the creation of robust programs Oracle (2021,). On the other hand, Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods, leading to more robust and reusable code.

To perform well in this laboratory and the next two, a good idea is to read Chapter 7: More Java Programming Concepts.

The first part of this lab aims to help you understand how to deal with different types of exceptions to prevent the application from crashing and offer user-friendly error messages. The second part aims to extend our understanding of Java generics, which allows us to write more generalized and type-safe code. We’ll also have some additional exercises touching on advanced features like lambda expressions, functional interfaces, and optional topics related to database handling and unit testing.

### 11.5.2 Exceptions

- New classes:
  - Sources
  - CourseManager – focusing on the changes made
  - CSVManager
  - SQLManager
  - XMLManager
  - MockClassesManager – focusing on the changes made
- Add exception handling to display potential errors in a user-friendly manner in the console.

The ”CSVManager” is going to read the information from comma separated value files specifically for each class. For example the Professor's corresponding CSV file will look like this:

| First Name | Last Name |
|---|---|
| John | Doe |
| Jane | Smith |
| Robert | Johnson |

Table 11.1: Sample Professor Data

The Course's CSV file will be something like this:

| **Name** | **Description** | **Teacher** | **Students** |
|---|---|---|---|
| Introduction to Programming | Basics of programming | John Doe | Alice Johnson; Bob Smith; Carol Brown |
| Algorithms | Advanced algorithms | Jane Smith | David Lee; Emma Davis; Frank Wilson |
| Web Development | Creating web apps | Robert Johnson | Grace Miller; Henry Taylor; Irene Anderson |
| Machine Learning | Fundamentals ML | Susan Brown | Jason Clark; Lisa White; Michael Turner |
| Database Management | managing data | William Davis | Olivia Baker; Peter Evans; Rachel Harris |

Table 11.2: Sample Course Data

### 11.5.3 Additional Exercises

- **Exception Handling Exercise:** Modify the CourseManager class to catch NullPointerException when trying to access a course that doesn't exist. Print a user-friendly error message.

- **Generic Types Exercise:** Modify the CourseManager class to work with a generic type T that extends Course. Test the new class with a subclass of Course.

- **Lambda Exercise for Sorting:** Use a lambda expression to sort the list of students in a course by their names alphabetically.

- **Functional Interface Exercise:** Define a functional interface with a method that takes two integers and returns an integer. Implement this using a lambda expression to perform addition, subtraction, multiplication, and division.

- **Database Exercise (optional):** Add exception handling around the database connection logic in SQLManager. Catch any SQL exceptions and print a user-friendly message.

- **Test Exercise (optional):** Write unit tests to verify that your exception handling is working as expected. For this, you might want to intentionally introduce errors (like null values or illegal arguments) and then catch them.

- **Advanced:** Introduce a logging framework like Log4J or SLF4J and replace all the System.out.println() statements with proper logging.

## 11.6 Laboratory 6: I/O operations

### 11.6.1 Introduction

Input/Output (I/O) operations are fundamental to any application that needs to interact with the external world, whether it's reading from or writing to files, databases, or other data streams Friesen (2015). Being able to effectively read from and write to files is particularly important for the long-term storage and retrieval of data. In Java, there are various ways to perform I/O operations, including using streams and readers/writers.

In this lab, we will focus on file I/O in Java. We'll take a practical approach by reading and writing student information to and from CSV files. We will modify the existing 'CourseManagerCSV' class to include methods that allow you to populate course, student, and professor information from files, as well as save this information back to files. By the end of this lab, you should have a good understanding of basic file I/O operations in Java and how to integrate them into larger projects.

### 11.6.2 Example of Reading/Writing to a File

The Class that will be used as a sample to read/write from/to a file:

```
class Student {
    String name;
    String surname;
    int group;

    public Student(String name, String surname, int group) {
        this.name = name;
        this.surname = surname;
        this.group = group;
    }

    public String toString() {
        return name + ", " + surname + ", " + group;
    }
}
```

#### Reading

```

static ArrayList<Student> studenti = new ArrayList<Student>();

static void readFromFile(String filepath) {
    try {
        File f = new File(filepath);
        BufferedReader br = new BufferedReader(new FileReader(f));
        String line = br.readLine();
        // Ignore the first line (header)
        if (line != null) {
            line = br.readLine();
        }
        while (line != null) {
            String[] tokens = line.split(",");
            Student s = new Student(tokens[0], tokens[1].trim(), Integer.
   parseInt(tokens[2].trim()));
            students.add(s);
            line = br.readLine();
        }
    } catch (Exception ex) {
        System.out.println(ex);
    }
}
```

#### Writing

```
static void writeToFile(String filepath) {
    try {
```

```
        File f = new File(filepath);
        BufferedWriter bw = new BufferedWriter(new FileWriter(f));
        try {
            students.add(new Student("Ionescu", "Dan", 3));
            bw.write("name, surname, group\r\n"); // write the header
            for (Student stud : students) {
                bw.write(stud.toString() + "\r\n");
            }
            bw.flush();
        } catch (IOException e) {
            System.out.println(e);
        } finally {
            bw.close();
        }
    } catch (Exception ex) {
        System.out.println(ex);
    }
}
```

### 11.6.3 Exercises

#### ManagerCourses

Modify the `CourseManagerCSV.java` file in the project as follows:

```
public class CourseManagerCSV extends CourseManager {
    File students, professors, courses;

    public CourseManagerCSV() {
        try {
            students = new File("students.csv");
            professors = new File("professors.csv");
            courses = new File("courses.csv");
        } catch (Exception ex) {
            System.out.println(ex);
        }
    }

    public CourseManagerCSV(File students, File professors, File courses) {
        this.students = students;
        this.professors = professors;
        this.courses = courses;
    }

    /* The function will bring data about students, professors, and courses
       from files and populate the collection in the Manager class: list */
    public void ReadDataFromFiles() {
        try {
            ArrayList<Student> studentList = PopulateStudents();
            ArrayList<Professor> profList = PopulateProfessors();
            ArrayList<Professor> coursesList = PopulateCourses();
```

```
27              //copy the lists to the coresponding members of the class
28          } catch (Exception ex) {
29              System.out.println(ex);
30          }
31      }
32      //...
33  }
```

**Tasks**

- Modify the `CourseManagerCSV.java` class in the project:
  - Implement the `PopulateStudents`, `PopulateProfessors`, and `PopulateCourses` methods.
  - Add a method to save the added or modified courses, called `WriteCoursesToFile`.
  - If a new professor is added, save them in `professors.csv`.
  - If a new student is added, save them in `students.csv`. If needed update the previous CSV files's structure.
  - Hint: The ID of a newly added student/professor is the ID of the last student/professor in the list + 1.
  - Complete the Add/Update/Delete methods for the student/professor/course. These methods should be part of the `CourseManagerCSV` class.

## 11.7 Laboratory 7: Threads

### 11.7.1 Introduction to Threads

In Java, a *Thread* is the path of execution for a program. Java is a multi-threaded programming language, which means we can develop multi-threaded programs using Java Goetz et al. (2006). A multi-threaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Threads can perform tasks independently and can also share resources to collaborate on a task.

### 11.7.2 Thread

- Calculate the sum of the first 1000 natural numbers using two separate threads (other than main) as follows:
  - The first thread works with the first 500 numbers, the second thread works with numbers greater than 500.
  - Each thread "requests" the next number to add to its own sum. Note: a number cannot be used by both threads.

- Main will control the output of the two threads.

### 11.7.3 Exercises

1. Reading from the files from the previous lab should be done on a thread separate from the main thread.
2. Writing to the files from the previous lab should also be done on a thread separate from the main thread.
3. Implement a way for the two threads to signal to the main thread that they have completed their tasks.
4. Modify the program to work with an arbitrary number of threads. Divide the sum task accordingly.
5. Use Java's synchronized blocks or methods to ensure that the threads do not use the same numbers.
6. Add a timer to measure how much time it takes for all threads to complete the task.

## 11.8 Laboratory 8: File I/O with JavaFX

### 11.8.1 Introduction

In this laboratory, we move a step further into real-world applications by integrating file I/O operations with a graphical user interface (GUI) using JavaFX Deitel and Deitel (2015); Oracle (2021,). Here, we will build upon the previous laboratory exercise by adding functionality to add new courses, professors, and students via a JavaFX application. A prerequisite for this laboratory is to read Chapter 9: Java Application Development. The laboratory aims to demonstrate how to read from and write to files through a user-friendly interface.

### 11.8.2 Functional Requirements

The JavaFX application should offer the following functionalities:

- Add a new course.
- Add a new professor.
- Display a list view containing the data read from the following three files:
  - Students
  - Professors
  - Courses

- Save any new additions to their respective files.
- Gracefully close the application window.

### 11.8.3 Exercises

#### JavaFX-Based Editing and Saving Application

Design an application using JavaFX that, in addition to viewing and adding new entries, allows for:

- Editing current students.
- Editing current professors.
- Editing current courses.

Remember to include functionality to save any changes made.

#### Improvements

Any additional improvements to enhance the functionality or user experience are welcome.

## 11.9 Laboratory 9: H2 Database and Spring Framework

### 11.9.1 Introduction

In this laboratory, we will elevate our data management capabilities by introducing an H2 in-memory database and utilizing the Spring Framework for Java. This laboratory will help you understand how to architect applications with greater scalability and maintainability.

A prerequisite for this laboratory and the next, is to read Chapter 10: Java Tools and Libraries.

### 11.9.2 Functional Requirements

The Java application should offer the following functionalities with the H2 database and Spring Framework:

- CRUD operations (Create, Read, Update, Delete) for courses, professors, and students.
- Display a list view for the courses, professors, and students.
- Implement transactions to ensure data consistency.
- Implement Spring Data JPA Repositories for data access.

### 11.9.3 Technology Stack

For this laboratory, you are expected to be familiar with:

- Java Programming Language
- H2 Database
- Spring Boot
- Spring Data JPA

### 11.9.4 Exercises

#### Database Configuration

Configure the H2 in-memory database with Spring Boot. Ensure that it is accessible via the Spring application.

#### Spring Data Repositories

Implement Spring Data JPA repositories for student, professor, and course entities.

#### CRUD Operations

Implement CRUD operations using the Spring Framework for managing students, professors, and courses.

#### Transactional Logic

Implement transactional logic to ensure that operations like adding a student to a course are atomic and consistent.

#### Display Data

Integrate your previous JavaFX user interface to display the data managed by the Spring application.

#### Improvements

Any additional features that can improve the application's functionality or user experience are welcome.

## 11.10 Laboratory 10: Networking, Messaging, and Apache Frameworks

### 11.10.1 Introduction

In this advanced laboratory, we will extend our Java application with real-time capabilities using networking protocols and messaging frameworks. We'll cover topics like Java Sockets for direct communication between applications, RabbitMQ for message queuing, and introduce various Apache frameworks that can augment your application's capabilities rab (2021).

### 11.10.2 Functional Requirements

The Java application should offer the following functionalities:

- Real-time updates between multiple instances of the application.
- Use Java Sockets for simple two-way communication.
- Use RabbitMQ for asynchronous messaging.
- Implement Apache Kafka for real-time data streaming.
- Introduce Apache Camel for routing data between different systems.

### 11.10.3 Technology Stack

For this laboratory, you are expected to be familiar with:

- Java Programming Language
- Java Sockets
- RabbitMQ
- Apache Kafka
- Apache Camel

### 11.10.4 Exercises

#### Java Sockets

Implement a simple chat feature between two instances of the Java application using Java Sockets.

### RabbitMQ

Add asynchronous messaging capabilities to your application. For instance, when a new course is added in one instance, a message should be sent to a queue and picked up by other instances to update their course lists.

### Apache Kafka

Implement a real-time data streaming solution using Apache Kafka. This could be useful for real-time analytics of data like student grades or course enrollments.

### Apache Camel

Use Apache Camel to route data from your application to other systems. For instance, automatically send an email to a student when they are enrolled in a new course.

### Additional Tasks

- Integrate your JavaFX interface for better interaction and real-time updates.
- Add any other features or frameworks you think would benefit your application.

# Bibliography


[1] 2021. Rabbitmq official getting started guide.

[2] J. Bloch. 2017. *Effective Java*. Pearson Education.

[3] Joshua Bloch. 2018. *Effective Java*. Addison-Wesley Professional.

[4] E. Burnette. 2005. *Eclipse IDE Pocket Guide: Using the Full-Featured IDE*. O'Reilly Media.

[5] Paul J Deitel and Harvey Deitel. 2015. *Java: How to Program, Early Objects*. Pearson.

[6] D.K. Fields, S. Saunders, and E. Belyaev. 2006. *IntelliJ IDEA in Action: Covers IDEA V.5*. In Action. Manning.

[7] Jeff Friesen. 2015. *Java I/O, NIO, and NIO.2*. Apress.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-wesley.

[9] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice*. Addison-Wesley Professional.

[10] E.R. Harold. 2000. *Java Network Programming*. Java (o'Reilly) Series. O'Reilly.

[11] Gosling J. 2018. *The Java® Language Specification Java SE 11 Edition*. Oracle America, Inc.

[12] M. Naftalin. 2007. *Java Generics and Collections*. Java Series, O'Reilly.

[13] Scott Oaks. 2014. *Java Performance: The Definitive Guide*, 1st edition. O'Reilly Media, Inc.

[14] Oracle. 2021. The java tutorials: Collections framework.

[15] Oracle. 2021. The java tutorials: Exceptions.

[16] Oracle. 2021. The java tutorials: Generics.

[17] Oracle. 2021. Javafx documentation project.

[18] H. Schildt. 2021. *Java: The Complete Reference, Twelfth Edition*. McGraw-Hill Education.

[19] J. Vos, S. Chin, W. Gao, J. Weaver, and D. Iverson. 2017. *Pro JavaFX 9: A Definitive Guide to Building Desktop, Mobile, and Embedded Java Clients*. Apress.

[20] C. Walls. 2016. *Spring Boot in Action*. Manning.

[21] Mark Allen Weiss. 2012. *Data Structures and Algorithm Analysis in Java.* Pearson Education.

[22] G. Wielenga. 2015. *Beginning NetBeans IDE: For Java Developers*. Apress.