📓 Java Learning Journey

Chapter 10 - Object-Oriented Thinking

✍️ **Prepared by:** Ahmed Elsifi

1. Class Abstraction and Encapsulation

- **Abstraction**
  - Think of it as *what something does* without worrying about *how it does it*.
  - For example: When you drive a car, you use the steering wheel, pedals, and gearshift. You don't need to know how the engine works inside—that's abstraction.
  - In programming, abstraction means defining a class's **interface (public methods, constructors, fields)** that others can use, while hiding its internal logic.
- **Encapsulation**
  - This means "wrapping" the data (fields/variables) and behavior (methods) inside one unit (the class) and **restricting direct access** to the data.
  - Example: You make a field `private` and provide `getters` and `setters` to control how it's accessed or modified.
  - This prevents outside code from messing with the class internals in an unsafe way.

👉 **Difference**:

- Abstraction = *Hiding unnecessary details, showing only the important parts.*
- Encapsulation = *Protecting the internal details from direct access.*

---

2. Procedural vs. Object-Oriented Paradigms

- **Procedural Programming**
  - Focuses on **functions/methods** (procedures).
  - Data and functions are separate. Functions "take in" data and perform operations.
  - Example: In C, you write functions like `add(x, y)` and keep data as global variables or structs.
- **Object-Oriented Programming (OOP)**
  - Data (fields) and behavior (methods) are **combined together** inside objects.
  - You model things as **objects** that represent real-world entities.
  - Example: A `Car` object has data (speed, color, fuel level) and behavior (drive(), brake(), refuel()).

👉 OOP makes programs more modular, reusable, and easier to maintain.

3. Class Relationships

- **Association**
  - A general connection between two classes.
  - Example: `Student` takes a `Course`. They are related, but each can exist independently.
- **Aggregation** (a weaker form of "has-a")
  - One class contains another, but the contained class **can still exist on its own**.
  - Example: A `Student` has an `Address`. Even if the `Student` object is deleted, the `Address` object might still exist elsewhere.
- **Composition** (a stronger form of "has-a")
  - The contained object **cannot exist without** the parent object.
  - Example: A `Student` has a `Name`. If the `Student` is destroyed, the `Name` is destroyed too.
- **Inheritance** ("is-a" relationship)
  - A child class derives from a parent class and **inherits fields and methods**.
  - Example: A `Dog` is an `Animal`.
  - This allows code reuse and specialization.

---

4. Wrapper Classes

- Java provides **wrapper classes** to use primitive data types as objects.
- Classes:
  - `Byte`, `Short`, `Integer`, `Long` → wrap integer types
  - `Float`, `Double` → wrap decimal types
  - `Character` → wraps a char
  - `Boolean` → wraps a boolean
- **Why use them?**
  - Collections like `ArrayList` work only with objects, not primitives. Wrappers solve this.
  - They provide useful methods, e.g. `Integer.parseInt("123")`.
- **Immutable**: Once created, wrapper objects cannot be changed.
- **valueOf()**
  - Example: `Integer.valueOf(5)`
  - Reuses cached objects for small values (–128 to 127), saving memory and improving performance.

5. Autoboxing and Auto-Unboxing

- **Autoboxing**
  - Automatic conversion from a primitive to its wrapper.
  - Example: `Integer x = 5;` → compiler converts it to `Integer.valueOf(5)`.
- **Auto-unboxing**
  - Automatic conversion from a wrapper to a primitive.
  - Example: `int y = x;` → compiler converts it to `x.intValue()`.

👉 These make code cleaner, but can cause **NullPointerException** if a null wrapper is unboxed.

---

6. BigInteger and BigDecimal

- **BigInteger**
  - Used when integers are too big for `long`.
  - Example: Cryptography (working with numbers bigger than 2^63–1).
- **BigDecimal**
  - Used for **precise decimal calculations** (like money or scientific values).
  - Example: `0.1 + 0.2` using `double` may give `0.3000000004`, but `BigDecimal` gives exact `0.3`.
- Both are **immutable**.
- They support operations like `add()`, `subtract()`, `multiply()`, `divide()`.

---

7. String Class

- **Immutable**
  - Once created, a String cannot be changed.
  - Example: `String s = "Hello"; s.concat("World");` → creates a new string `"HelloWorld"` instead of modifying `s`.
- **String Interning**
  - Java reuses string literals to save memory.
  - Example: Two `"Hello"` literals point to the same memory location in the string pool.
- **Methods**
  - `length()`, `charAt()`, `substring()`, `indexOf()`, `equals()`, `compareTo()`, `matches()` (for regex).

👉 Strings are perfect when text doesn't change much.

8. StringBuilder and StringBuffer

- Both are **mutable string classes** (unlike String).
- They allow you to modify text **without creating new objects** every time.
- **StringBuffer**
  - Thread-safe (synchronized).
  - Slower, but safe to use in multi-threaded environments.
- **StringBuilder**
  - Not synchronized (not thread-safe).
  - Faster in single-threaded code (most cases).
- Common methods: `append()`, `insert()`, `delete()`, `reverse()`, `replace()`.

👉 Use `StringBuilder` for speed unless you specifically need thread safety.

👉 Thread-safe means the class or method ensures correct results when accessed by multiple threads at the same time.

🔐 Access Modifiers in Java

Java provides four access modifiers to control the visibility and accessibility of classes, methods, and fields:

Modifier	Class	Package	Subclass	World
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
`default`	✅	✅	❌	❌
`private`	✅	❌	❌	❌

1. `public`

- Accessible everywhere. Example: `public int count;`

2. `protected`

- Accessible in same package + subclasses. Example: `protected void calculate() {}`

3. `default` (no modifier)

- Accessible only in same package. Example: `int value;`

4. `private`

- Accessible only in same class. Example: `private String secretData;`

📚 Essential Java Libraries & Imports

1. Wrapper Classes & String Class

```
import java.lang.*; // Auto-imported (no need to explicitly import)
// Includes: String, Integer, Double, Boolean, Character, etc.
```

2. BigInteger & BigDecimal

```
import java.math.BigInteger;
import java.math.BigDecimal;
```

3. Scanner for Input

```
import java.util.Scanner;
```

🎁 Wrapper Classes: Why Do We Use Them?

Purpose:

1. **To treat primitives as objects** (required by collections like ArrayList)
2. **To use utility methods** for conversion, comparison, and parsing
3. **To support null values** (primitives cannot be null)

List of Wrapper Classes:

- `Byte` - wraps `byte`
- `Short` - wraps `short`
- `Integer` - wraps `int`
- `Long` - wraps `long`
- `Float` - wraps `float`
- `Double` - wraps `double`
- `Character` - wraps `char`
- `Boolean` - wraps `boolean`

Key Characteristics:

- **Immutable**: Once created, their values cannot change
- **Contain useful constants**: `MIN_VALUE`, `MAX_VALUE`
- **Provide conversion methods**: `parseInt()`, `valueOf()`, `toString()`

🔢 Parsing Strings to Numbers

Each wrapper class provides methods to convert strings to primitive values:

Common Parsing Methods:

Method	Example	Returns
`Integer.parseInt(String s)`	`Integer.parseInt("123")`	`int`
`Double.parseDouble(String s)`	`Double.parseDouble("3.14")`	`double`
`Boolean.parseBoolean(String s)`	`Boolean.parseBoolean("true")`	`boolean`
`Long.parseLong(String s)`	`Long.parseLong("1000000")`	`long`

With Radix (Number Base):

```java
int decimal = Integer.parseInt("10"); // 10
int binary = Integer.parseInt("1010", 2); // 10 (in decimal)
int hex = Integer.parseInt("A", 16); // 10 (in decimal)
```

ValueOf() vs ParseXXX():

- `parseInt()` returns primitive `int`
- `valueOf()` returns `Integer` object

🔢 BigInteger Class

Purpose:

Handle integers of any size (beyond `long` range)

Key Methods:

Method	Description
`add(BigInteger val)`	Returns sum
`subtract(BigInteger val)`	Returns difference
`multiply(BigInteger val)`	Returns product
`divide(BigInteger val)`	Returns quotient
`remainder(BigInteger val)`	Returns remainder
`compareTo(BigInteger val)`	Compares values

Example:

```
BigInteger a = new BigInteger("12345678901234567890");
BigInteger b = new BigInteger("98765432109876543210");
BigInteger c = a.multiply(b); // Very large number
```

🔢 BigDecimal Class

Purpose:

Precise decimal calculations (avoid floating-point errors)

Key Methods:

Method	Description
`add(BigDecimal val)`	Returns sum
`subtract(BigDecimal val)`	Returns difference
`multiply(BigDecimal val)`	Returns product
`divide(BigDecimal val, int scale, RoundingMode mode)`	Returns quotient with precision control
`setScale(int newScale, RoundingMode mode)`	Sets scale with rounding

Example:

```
BigDecimal a = new BigDecimal("1.05");
BigDecimal b = new BigDecimal("2.30");
BigDecimal c = a.multiply(b); // Precise: 2.4150
```

Important:

Always use `String` constructor for predictable results:

```
// Prefer this:
BigDecimal exact = new BigDecimal("1.05");

// Over this (may have precision issues):
BigDecimal approx = new BigDecimal(1.05);
```

📝 String vs StringBuilder vs StringBuffer

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread Safety	Not applicable	Not thread-safe	Thread-safe
Performance	Fast for read	Fast for modification	Slower due to synchronization
When to Use	When value won't change	Single-threaded environment	Multi-threaded environment

🔍 Regular Expression (Regex) Quick Reference

Pattern	Meaning	Example
`.`	Any character	`"a.c"` matches "abc", "a c"
`\d`	Digit	`"\d\d"` matches "12", "99"
`\D`	Non-digit	`"\D\D"` matches "ab", "#$"
`\w`	Word character	`"\w+"` matches "hello", "java123"
`\W`	Non-word character	`"\W"` matches "@", "!"
`\s`	Whitespace	`"a\sb"` matches "a b"
`\S`	Non-whitespace	`"\S\S"` matches "ab", "#$"
`[abc]`	Any of a, b, or c	`"[aeiou]"` matches "a", "e"
`[^abc]`	Not a, b, or c	`"[^0-9]"` matches "a", "!"
`*`	Zero or more	`"a*b"` matches "b", "ab", "aab"
`+`	One or more	`"a+b"` matches "ab", "aab"
`?`	Zero or one	`"a?b"` matches "b", "ab"
`{n}`	Exactly n times	`"a{3}"` matches "aaa"
`{n,}`	At least n times	`"a{2,}"` matches "aa", "aaa"
`{n,m}`	Between n and m times	`"a{2,4}"` matches "aa", "aaa"

📋 Methods Tables

🔤 String Class Methods

Method	Return Type	Description
`charAt(int index)`	`char`	Returns char at specified index.
`length()`	`int`	Returns length of string.
`substring(int begin)`	`String`	Returns substring from begin index to end.
`substring(int begin, int end)`	`String`	Returns substring from begin to end-1.
`concat(String str)`	`String`	Concatenates specified string.
`replace(char old, char new)`	`String`	Replaces all occurrences of old char with new.
`replace(CharSequence old, CharSequence new)`	`String`	Replaces all occurrences of old sequence with new.
`replaceAll(String regex, String replacement)`	`String`	Replaces all substrings matching regex.
`replaceFirst(String regex, String replacement)`	`String`	Replaces first substring matching regex.
`toLowerCase()`	`String`	Converts to lowercase.
`toUpperCase()`	`String`	Converts to uppercase.
`trim()`	`String`	Removes leading and trailing whitespace.
`split(String regex)`	`String[]`	Splits string around matches of regex.
`equals(Object obj)`	`boolean`	Compares string to another object.
`equalsIgnoreCase(String another)`	`boolean`	Compares ignoring case.
`compareTo(String another)`	`int`	Lexicographical comparison.
`indexOf(int ch)`	`int`	Returns index of first occurrence of char.
`lastIndexOf(int ch)`	`int`	Returns index of last occurrence of char.
`valueOf(...)`	`String`	Static method to convert various types to string.

Method	Return Type	Description
`format(String format, Object... args)`	`String`	Returns formatted string.
`matches(String regex)`	`boolean`	Tells if string matches regex.
`toCharArray()`	`char[]`	Converts string to char array.
`getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`	`void`	Copies chars to destination array.

🛠 StringBuilder and StringBuffer Methods

Method	Return Type	Description
`append(...)`	`StringBuilder`	Appends data (various types).
`insert(int offset, ...)`	`StringBuilder`	Inserts data at specified position.
`delete(int start, int end)`	`StringBuilder`	Deletes subsequence.
`deleteCharAt(int index)`	`StringBuilder`	Deletes char at index.
`replace(int start, int end, String str)`	`StringBuilder`	Replaces subsequence with string.
`reverse()`	`StringBuilder`	Reverses sequence.
`setCharAt(int index, char ch)`	`void`	Sets char at index.
`charAt(int index)`	`char`	Returns char at index.
`length()`	`int`	Returns length.
`capacity()`	`int`	Returns current capacity.
`setLength(int newLength)`	`void`	Sets length of sequence.
`substring(int start)`	`String`	Returns substring from start.
`substring(int start, int end)`	`String`	Returns substring from start to end-1.
`toString()`	`String`	Returns string representation.
`trimToSize()`	`void`	Trims capacity to current length.

🧩 Regular Expressions (Regex) Rules

Pattern	Meaning
`.`	Any single character
`*`	Zero or more occurrences
`+`	One or more occurrences
`?`	Zero or one occurrence
`{n}`	Exactly n times
`{n,}`	At least n times
`{n,m}`	Between n and m times
`[...]`	Any one character in brackets
`[^...]`	Any one character not in brackets
`	OR operator
`^`	Beginning of line
`$`	End of line
`\d`	Digit [0-9]
`\D`	Non-digit
`\w`	Word character [a-zA-Z0-9_]
`\W`	Non-word character
`\s`	Whitespace
`\S`	Non-whitespace
`\b`	Word boundary
`\B`	Non-word boundary

💡 Important Notes

- Use `String` for immutable strings.
- Use `StringBuilder` for mutable strings in single-threaded environments.
- Use `StringBuffer` for mutable strings in multi-threaded environments.
- Prefer `valueOf()` over constructors for wrapper classes.
- Use `BigInteger` and `BigDecimal` for very large numbers or high precision.
- Regex is powerful for string matching, replacing, and splitting.