Advanced Java

by BPI

What we'll cover

1. Packages
2. Access Modifiers
3. Non-Access Modifiers
4. Data Types

Packages

- “A package is a namespace for organizing classes and interfaces in a logical manner”
- A way of bundling and *packaging* similar/related code
- Similar to file directories or folders
- Make classes easier to find and use, avoid naming conflicts, and control access
- Especially useful in large libraries where codes need to be organized and should be located in intuitive package names
- Failing to add a package puts your code in an unnamed package which is not advisable

Creating a Package

- Choose a name for your class's package
- At the top of your class, use the keyword "package" then, separated by a space, input the chosen name for the package
- Use lowercase to avoid conflicting with Class names
- Prepend the package with your company's domain in reverse order:
  - example: bpi.com.ph -> ph.com.bpi.training
- in cases wherein a the domain name begins with numbers or contains a hyphen (-), substitute with an underscore
  - example: 123bpi.com.ph -> ph.com._123bpi.training

Creating a Package

```
package act.one.lecture.model; // declaring this class's package

public class Patient extends BaseEntity {

    private Name name;
```

Using a Package Member

To call a *public* class member you can:

1. Refer to the member's fully qualified name
2. Import the package member
3. Import the member's entire package

Refer to the Member's Fully Qualified Name

- The fully qualified name of a member is the combination of it's package and it's member name
- It can be used directly in the code when it's package is not yet imported
- It not advisable to use this as it becomes tedious when you repeatedly call the member in your program

Refer to the Member's Fully Qualified Name

```
package act.one.lecture.model;


public abstract class BaseEntity {

    private Long id;

    // the attributes below use the fully qualified name
    private java.time.LocalDateTime createdDate;
    private java.time.LocalDateTime lastModifiedDate;
    private String createdBy;
    private String lastModifiedBy;

    public BaseEntity() {

        // example use of the fully qualified name
        this.setCreatedDate(java.time.LocalDateTime.now() );
        this.lastModifiedDate = java.time.LocalDateTime.now();
    }
```

Importing a Package Member

- using the keyword “import” followed by the package name and the member name allows that member to be referred to using their simple name

Importing a Package Member

```
package act.one.lecture.model;

import java.time.LocalDate;    // import the LocalDate class inside java.time

public class Patient extends BaseEntity {

    private Name name;
    private LocalDate birthdate;    // you can now use LocalDate as usual
```

Importing an Entire Package

- using the keyword “import” followed by the package name of your choice and an asterisk (*) as a wildcard character
- This allows all members under that package to be imported
- It cannot be used in conjunction with any other character
  - example: import java.time.L* // this is not valid

Importing an Entire Package

```
package act.one.lecture;

import java.time.*; // import the entire "Time" package members

public class PatientApplication {

    public void start() {

        LocalDateTime dateTime = LocalDateTime.now();    // use the simple name of classes
        LocalDate date = LocalDate.now();
    }
```

Package	What It Contains	Common Production Use Cases
java.lang	Core language classes (String, Math, Object, Thread, Enum, Throwable, etc.)	Everywhere. Strings, enums, wrappers, threads, exceptions
java.util	Collections (List, Map, Set), Iterator, Random, Optional, Scanner, Date/Calendar, UUID	Data structures, algorithms, utilities, configs, randomness
java.io	Input/Output streams, Readers, Writers, File, Serialization	File reading, writing, buffers, byte/char processing
java.nio / java.nio.file	Non-blocking I/O, Path, Files, Channels, ByteBuffer	Fast file operations, async I/O, memory buffers
java.time	Modern Date & Time API (LocalDate, Instant, Duration, ZoneId, DateTimeFormatter)	All date/time handling (replaces Date/Calendar)
java.math	BigDecimal, BigInteger, RoundingMode	Financial calculations, precise decimal math
java.net	Networking (URL, Socket, InetAddress, URI, HttpURLConnection)	HTTP calls (legacy), TCP/UDP sockets
java.sql	JDBC classes (Connection, ResultSet, PreparedStatement, SQLException)	Database access, JDBC templates, ORM integrations
java.util.concurrent	Executors, ThreadPool, Future, Locks, Concurrent collections	Multithreading, async tasks, thread-safe structures
java.util.regex	Pattern, Matcher	Validation, parsing, complex text extraction

Package	What It Contains	Practical Use
java.util.stream	Stream API, Collectors	Functional programming, pipelines, filtering, mapping
java.lang.reflect	Reflection API, Method, Field, Constructor	Frameworks, annotations, runtime inspection
java.util.logging	Built-in logging	Lightweight logging (rare in production—Log4j/SLF4J preferred)
java.security	Cryptography, Key management	Encryption, hashing, JWT signing, SSL
javax.net.ssl	SSL sockets, certificates	HTTPS, SSL handshake control
java.rmi	Remote Method Invocation	Rare; legacy distributed systems
javax.crypto	AES, RSA, ciphers, Mac	Secure data encryption
javax.annotation	@PostConstruct, @PreDestroy	Used heavily in Spring (Jakarta version now)

Access Modifiers

- Keywords that control the visibility and accessibility of classes, methods, and fields

Four Access Modifiers

1. Public
2. Private
3. Protected
4. Package-Private

Public

- Accessible from any class or package
- Widest visibility
- Used for classes, methods, or variables that are meant to be accessed publicly (with respect to the pillars of OOP)

Private

- Accessible only within the class
- Hide data that is not meant to be publicly accessible
- Or hide methods that contain internal logic

Protected

- Accessible within the subclass and classes within the same package.
- Used to provide limited visibility for subclasses
- More appropriate for inheritance as it allows subclasses to inherit the fields.

Package-Private

- If no keyword is provided, it will implicitly use ‘default’ access level. Accessible by classes within the same package

The importance of Access Modifiers

- Enforces “Encapsulation” where data is hidden and behavior is exposed
- Private attributes ensure that any update is done by the object itself and goes through the object itself
- Private attributes makes code loosely coupled; changes won’t break other parts of your code

The importance of Access Modifiers

- using setters and getters allow us to add code validation or any other logic before setting the value
- in the context of Threads, using setters allow us to add the keyword “synchronized” to the method, ensuring safe access to the value of the attribute
- Example: Show example of how code can break when we use public instead of private

Example:

```
Person p = new Person();
p.age = 25;
int x = p.age + 5;

// what if we in the future, requirements make us change
// int age  -> LocalDate birthDate
// above code will break and will need changing
```

```
// if we instead had the code below
private LocalDate birthdate;
public int getAge() {
   return Period.between(birthdate, LocalDate.now()).getYears();
}
public void setBirthdate(LocalDate bd) {
   this.birthdate = bd;
}

// and had:
Person p = new Person();
int x = p.getAge() + 5;
```

Non-Access Modifiers

- keywords that provide a characteristic to a class, attribute or method

Non-Access Modifiers

1. static
2. final
3. abstract
4. synchronized
5. volatile

static

- Belongs to the class, not to individual objects
- Shared by all instances
- Can be applied to variables, methods, nested classes
- Commonly used for utility methods, constants, shared counters, accessing members without creating an object
- Example:

final

- indicates that the variable, method, or class cannot be changed after assignment
- final + variable = the value cannot be changed
- final + method = the method cannot be overridden
- final + class = cannot be subclassed

abstract

- indicates that method or class should be implemented
- abstract + method = the method must be overridden
- abstract + class = cannot be instantiated, should be subclassed

synchronized

- ensures that the method or block of code can only be accessed by one thread

volatile

- indicates that the variable must always be read from the main memory
- ensures that the value is always updated and is visible to all other threads

Primitive Data Types

- Data type that is part of the programming language
- Dictates what kind of value is stored within the variable
- Stores the data itself
- Default value of 0 (depending on the data type)

Primitive Data Types

1. byte
2. short
3. int
4. long
5. float
6. double
7. char
8. boolean

Non-Primitive Data Types

- Stores references to the object, not the actual data.
- Examples are classes, interfaces, arrays
- Has default value of null

Wrapper Classes

- Classes that wrap around a primitive data type
- Allows primitive data types to be treated as objects, therefore allowing it to be used with classes that work with objects (collections) or allowing it to have the value of “null”.

Converting Data types

1. Implicit Type Conversion (Widening)
2. Explicit Type Conversion (Narrowing / Type Casting)
3. Using Conversion Methods (Wrapper Classes)

Implicit Type Conversion (Widening)

- Assign a larger (wider) data type to a smaller data type automatically converts it
- Example:
- > int myNumber = 10;
- > long myLongNumber = myNumber; // widened

Explicit Type Conversion (Narrowing / Type Casting)

- From a larger data type to a smaller data type
- Can result in data loss
- Example:
- > double myDecimal = 1.25
- > int myNumber = (int) myDecimal // value is now

Using Conversion Methods (Wrapper Classes)

- Use the methods provided by wrapper classes to convert data type
- Example:
- > String numberStr = "42";
- > int intValue = Integer.parseInt(numberStr); // Using Integer.parseInt to convert a String to int

Group Project Requirements

- Groups must create an Application that simulates a Library system
- Four important classes: Library, Book, Loan, User
- User is created at the start, give User a name (and/or id).
- A Book has an Author and a Title (and/or id).
- A Library can contain up to 5 Books.
- The User can borrow a Book or return a Book.
- A Loan is created when the User borrows a Book.
- A Loan is removed when the User returns a Book.
- A User can borrow up to 5 Books.
- The application must be able to: 1) Display ALL books, 2) Display AVAILABLE books, 3) Display BORROWED books (and who borrowed / Loan details)
- **Use of Java Collections is not allowed. (List, ArrayList, Map, HashMap, Set, HashSet, etc.)**
- **Java Exceptions + Handling is not allowed (Exception, try-catch)**
    - Up to the groups on how to implement 5 Books / 5 Loans
    - Added base template to follow, feel free to add new classes and add new attributes but DO NOT delete the existing classes.

Group Project Points System

- 10 Points if minimum requirements are done
- +1 Point if group is able to use Interface or Abstract Classes
- +1 Point Input Validation (No Exception Handling)
- +1 Point for clean implementation / elegant solution
- +1 Point for good formatting / display
- -2 Points for using public attributes / Failure to use getters and setters
- -2 Points for incorrect naming convention (variables = camelCase, Class = PascalCase)
- -2 Points for using advanced topics (Java collections / Java Exception handling, any out-of-scope or not discussed previously)

Group Project:

```
/*
 * 1. Upon application start, ask user to create one User
 * 2. Create one Library object
 * 3. Initialize 5 Book objects and add it to all Library slots
 * 4. Display options:
 *
 * - [1] Display All Books
 * - [2] Display Available Books
 * - [3] Display All Borrowed Books
 * - [4] Borrow Book
 * - [5] Return Book
 * - [6] Exit
 *
 * - user selects the number of the option
```

Group Project:

```
*   [1] Display All Books
* - Display all Books (ID, Title and Author) regardless if there is a Loan existing for
*
*   [2] Display Available Books
* - Display Books that do not have a Loan slot
*
*   [3] Display All Borrowed Books
* - Display Books that have a Loan equivalent.
* - Display the Book title and the User name of borrower
*
*   [4] Borrow Book
* - Displays all available books and User selects what book to borrow
* - Create a Loan object, set Loan id set Book and set User to current user
*
*   [5] Return Book
* - Display all Loans, user selects the Loan and removes that from the slot
*
*   [6] Exit
* - Stops the program
* */
```