Chapter 1: Introduction to Object Oriented Programming

1 Introduction

1.1 Object

The fundamental building block. A software object has:

- **State**: Represented by attributes (data) and relationships. The state can change over time.
- **Behavior**: Represented by operations or methods (functions that act on the object's data). Determines how the object responds to messages.
- **Identity**: Each object is unique, even if it has the same state as another object.

1.2 Class

A blueprint or template for creating objects. It defines:

- The common attributes and methods that all objects of that type will have.
- An object is an instance of a class.

1.3 Message Passing

Objects interact with each other by sending messages:

- A message is a request for an object to perform one of its operations.
- This is how behavior is invoked.

2 Object-Oriented Technology (OOT)

2.1 Definition

A collection of rules, languages, databases, and tools that are applied to build software. The core rules are:

- Inheritance
- Abstraction
- Encapsulation
- Polymorphism

2.2 Benefits

- **Reusability**: texttt and architectures can be reused.
- **Maintainability**: Changes are often localized to specific objects, making the system more stable.
- **Adaptability**: Easier to adapt to changing requirements.
- **Real-world Modeling**: Objects can represent real-world entities more naturally.

3 Object Interactions

3.1 Object-Oriented vs. Procedural

- **Procedural**: Focuses on procedures/functions; data is separate.
- **Object-Oriented**: Focuses on objects; data and methods are encapsulated. A program is a set of objects interacting via messages.

4 Java-Specific Concepts

4.1 Java Platform

Provides the environment for running Java programs. Key components are:

- **Java Virtual Machine (JVM)**: Executes Java bytetexttt, providing platform independence ("Write Once, Run Everywhere"). Handles memory management (garbage collection).
- **Java API (Application Programming Interface)**: A large library of pre-written classes and interfaces for common tasks.

4.2 Java Editions

- JSE: Java Standard Edition
- JEE: Java Enterprise Edition
- JME: Java Micro Edition
- Java Card

4.3 Java Compilation Model

- Source texttt (.java) is compiled into bytetexttt (.class).
- Bytetexttt is platform-independent and is executed by the JVM.
- The JVM uses a Just-In-Time (JIT) compiler to translate bytetexttt into native machine texttt for the specific CPU.

4.4 Java Features

- Object-Oriented
- Simple (relatively, compared to some other languages)
- Platform Independent (due to the JVM)
- Network-Capable (designed for distributed applications)
- Multi-threaded (supports concurrent execution)
- Robust (memory management, exception handling)
- Large Standard Library

4.5 Java Application Types

- **Application**: Standalone programs that run directly on the JVM (not in a browser). The `main()` method is the entry point.
- **Applet**: GUI applications that run within a web browser (requires a JVM plugin). Less common now due to security concerns and the rise of other web technologies.
- **Web Application**: Dynamic content on the server.

5 The Problem: Why Version Control?

5.1 Individual Developer Pain Points

- **Accidental breakage**: Making a small change that unexpectedly breaks working texttt, and being unable to easily revert.
- **Lost work/inability to revert**: Having a working version, making changes that don't work, and needing to go back to the previous state (but not having it).
- **Difficulty tracking changes**: Not knowing exactly what changes were made between working and non-working versions.

5.2 Team Collaboration Pain Points

- **Conflicting changes**: Multiple developers working on the same file(s) simultaneously, leading to overwrites or integration problems.
- **Merging difficulties**: Combining changes made by different developers to the same files.
- **Tracking responsibility**: Knowing who made which changes and why.

6 Version Control Systems (VCS) - The Solution

6.1 Definition

A system that records changes to a file or set of files over time so that you can recall specific versions later. Also called Source texttt Control (SCC) systems.

6.2 Key Capabilities

- **Version History**: Keeps track of all versions of files, not just the latest.
- **Change Tracking**: Records what changed, who changed it, and why (through commit messages).
- **Collaboration Support**: Provides mechanisms for multiple developers to work on the same project without overwriting each other's changes.
- **Branching and Merging**: Allows for parallel development efforts and the ability to combine changes from different branches.
- **Reverting**: Enables going back to previous versions of files or the entire project.

7 Versioning Models

7.1 Lock-Modify-Unlock (Pessimistic Locking)

- **Mechanism**: Only one developer can modify a file at a time. A developer must "lock" the file before making changes, preventing others from editing it until the lock is released.
- **Analogy**: Like a library book that can only be checked out by one person at a time.
- **Pros**: Prevents conflicts by ensuring exclusive access.
- **Cons**: Can slow down development if developers frequently need to wait for locks to be released. Can lead to bottlenecks.

7.2 Copy-Modify-Merge (Optimistic Locking)

- **Mechanism**: Developers work on their own copies of files. When they're ready to integrate their changes, they "commit" them. The VCS attempts to merge the changes automatically. If conflicts are detected (multiple developers changed the same lines), manual resolution is required.
- **Analogy**: Like multiple people editing copies of a document, then combining the changes later.
- **Pros**: Allows for more parallel development. More efficient when conflicts are infrequent.
- **Cons**: Requires conflict resolution when simultaneous changes occur.

7.3 Distributed Version Control (DVCS) (e.g., Git, Mercurial)

- **Mechanism**: Each developer has a complete copy of the entire repository, including the full history. Changes are made locally, then "pushed" to a central repository (or shared directly with other developers). Multiple repositories can exist.
- **Analogy**: Like each developer having their own personal library with all the books, and occasionally synchronizing with a central library.

- **Pros**: Highly flexible. Supports offline work. Fast local operations. Robust (no single point of failure). Encourages branching and merging.
- **Cons**: Can be more complex to understand initially. Requires careful management of multiple repositories.
- **Compared to Central VCS**: Central has one repository, all operations depend on it.

8 Key Vocabulary

- **Repository**: The central storage location for all files and their history (the "database" of the VCS).
- **Working Copy/Local Repository**: A developer's personal copy of the files from the repository.
- **Checkout/Clone**: The process of getting a working copy from the repository. `clone` (in DVCS) copies the entire repository; `checkout` (in centralized VCS) typically gets only the latest version.
- **Commit/Check-in**: The process of saving changes from the working copy to the repository, creating a new version.
- **Update/Pull/Fetch**: The process of getting the latest changes from the repository into the working copy. `pull` (and `fetch` + merge) combines getting changes and integrating them.
- **Merge**: The process of combining changes from different versions or branches.
- **Conflict**: Occurs when multiple developers have made changes to the same lines of a file, and the VCS cannot automatically resolve the differences. Requires manual intervention.
- **Revert/Undo Changes**: Discarding local changes and going back to a previous version from the repository.
- **Branch**: A separate line of development, allowing for parallel work on features or bug fixes without affecting the main texttttbase.
- **Tag/Label**: A named snapshot of a specific version of the project (often used to mark releases).
- **Change/Change Set/Change List**: List of changes before committing.

9 Tools

9.1 Diff Tools

- Show differences among files.

9.2 Version Control Systems

- Tools for performing version control.
- **Git/GitHub/GitLabs/Docker**: A popular, free, open-source, distributed version control system.

Chapter 2: UML & Use Case Diagram and Java Basic

1 Modeling and its Importance

1.1 What is a Model?

A simplification of reality. It helps us understand complex systems by focusing on essential aspects and ignoring unnecessary details.

1.2 Why Model?

- **Visualization**: Helps to see the system before it's built.
- **Specification**: Defines the structure and behavior of the system.
- **Construction Template**: Provides a blueprint for building the system.
- **Documentation**: Records design decisions.
- **Understanding**: Breaks down complex systems into manageable parts.

1.3 The Problem of Not Modeling

Teams that don't model often end up with poorly designed systems, experience delays, and face a higher risk of failure. Modeling is crucial for successful, complex projects.

2 UML (Unified Modeling Language)

2.1 What is UML?

A standardized visual language for specifying, visualizing, constructing, and documenting the artifacts of software systems (and other systems, too). It's not a methodology, but a notation.

2.2 Why UML?

Provides a common language for software developers, analysts, and stakeholders to communicate about system design. Unifies various object-oriented methods that existed before its standardization.

2.3 Key Aspects of UML

- **Visualizing**: Uses diagrams to represent different aspects of a system.
- **Specifying**: Creates precise and unambiguous models.
- **Constructing**: Can be linked to programming languages (texttt generation - forward engineering) and can be created from existing texttt (reverse engineering).
- **Documenting**: Provides a way to document requirements, architecture, tests, and project plans.

3 Requirements and Software Behavior

3.1 Purpose of Requirements

- Define what the software should do (agreement with stakeholders).
- Help developers understand the software's purpose.
- Define the scope of the software.
- Provide a basis for planning and estimation.
- Define the user interface.

3.2 Software Behavior

How the software acts and reacts to interactions. It encompasses the actions and activities of the software.

4 Use Cases and Use Case Diagrams

4.1 Use Case

A description of a sequence of actions that a system performs, yielding an observable result of value to a particular actor. It:

- Represents a specific way of using the system.
- Models a dialogue between actors and the system.
- Focuses on what the system does, not how it does it.

4.2 Benefits of Use Cases

- **Communication**: Easy for users and developers to discuss.
- **Identification**: Clearly identify actors and features.
- **Verification**

4.3 Actor

Represents a role that a user (or another system) plays when interacting with the system. Actors are external to the system. An actor can be a person, a device, or another system.

- **Guideline for identifying actors**: Find the subject of actions.

4.4 Use Case Diagram

A visual representation of the relationships between actors and use cases. It:

- Shows which actors interact with which use cases.
- Provides a high-level overview of the system's functionality.
- **Guideline for identifying use cases**: Find the series of actions.

4.5 Key Elements of a Use Case Diagram

- **Actor**: Represented by a stick figure.
- **Use Case**: Represented by an oval.
- **Association**: A line connecting an actor and a use case, indicating that the actor participates in the use case. An optional arrowhead on the association can indicate who initiates the interaction.

5 Identifiers

5.1 Definition

Names used to represent variables, methods, classes, and labels in Java texttt.

5.2 Rules

- Must start with a letter, `$`, or _.
- Can contain letters, digits, `$`, and _.
- Cannot be a Java keyword (e.g., `int`, `class`, `if`, `for`).
- Case-sensitive (`myVariable` is different from `MyVariable`).

5.3 Convention

- **Package**: All in lowercase.
- **Class**: Start with an upper-case letter.
- **Method/Field**: Start with a lower-case letter, and then the first letter of each remaining word is in upper-case.
- **Constants**: All in upper-case letters.

5.4 Literals

Represent fixed values in the texttt (e.g., `10`, `3.14`, `true`, `"hello"`).

6 Data Types

6.1 Two Main Categories

- **Primitive Types**: Represent basic data values.
- **Reference Types**: Represent references (pointers) to objects.

6.2 Primitive Types

Java has four categories of primitive types:

- **Integer**: `byte`, `short`, `int`, `long` (whole numbers, signed). Different sizes and ranges.
- **Floating-Point**: `float`, `double` (numbers with decimal points). Different sizes, ranges, and precision.
- **Character**: `char` (represents a single Unitexttt character).
- **Boolean**: `boolean` (represents `true` or `false`).

6.3 Reference Types

- **Array**: A collection of elements of the same type.
- **Object**: An instance of a class.
- **Class, Enum, Interface**.

6.4 Literals (Examples)

- **Integer**: `10`, `-5`, `07` (octal), `0x1A` (hexadecimal), `26L` (long).
- **Floating-Point**: `3.14`, `2.7f` (float), `6.02e23` (scientific notation).
- **Boolean**: `true`, `false`.
- **Character**: `'a'`, `'%'`, `'\n'` (newline - escape sequence).
- **String**: `"Hello, world!"`.

6.5 Escape Sequences

Special character combinations.

6.6 Casting

- **Implicit Cast**: From a smaller size to a larger size data.
- **Explicit Cast**: Cast a larger size data to a smaller one.

7 Operators

7.1 Definition

Symbols that perform operations on values (operands) to produce a result.

7.2 Categories

- **Arithmetic**: `+`, `-`, `*`, `/`, `%` (modulo - remainder).
- **Bitwise**: `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<<` (left shift), `>>` (right shift).
- **Relational**: `==` (equal to), `!=` (not equal to), `>`, `<`, `>=`, `<=`.
- **Logical**: `&&` (logical AND), `||` (logical OR), `!` (logical NOT).
- **Assignment**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, ≙, `<<=`, `>>=`.
- **Unary**: `+` (positive), `-` (negative), `++` (increment), `--` (decrement), `!` (logical NOT).

7.3 Operator Precedence

Determines the order in which operators are evaluated in an expression. Parentheses can be used to override the default precedence.

8 Control Statements

8.1 Purpose

Control the flow of execution in a program.

8.2 Types

- **if-else**: Conditional execution based on a boolean expression.

```
        if (condition)
        {
            // statements to execute if condition is true
        }
        else
        {
            // statements to execute if condition is false (optional)
        }
```

- **switch-case**: Multi-way branching based on the value of an expression.
  - Used for selecting one of several texttt blocks based on the value of a variable.
  - The variable must be an integer type, a `char`, a `String`, or an `enum`.

```
        switch (variable)
        {
            case value1:
                // statements
              break;
            case value2:
                // statements
              break;
                // ... more cases
            default:
                // statements (optional)
        }
```

- **while**: Repeats a block of texttt as long as a condition is true.

```
        while (condition)
        {
            // statements to repeat
        }
```

- **do-while**: Similar to while, but the texttt block is executed at least once before the condition is checked.

```
        do
        {
            // statements to repeat
        }
        while (condition);
```

- **for**: A more structured loop, often used for iterating a specific number of times.

```
        for (initialization; condition; increment/decrement)
        {
            // statements to repeat
        }
```

- **Loop Control Statements**
  - `break`: Exit a loop.
  - `continue`: Jump to the next step.

9 Arrays

9.1 Definition

A fixed-size, ordered collection of elements of the same data type.

9.2 Declaration and Instantiation

- `dataType[] arrayName = new dataType[arraySize];` (e.g., `int[] numbers = new int[5];`)
- `dataType arrayName[] = new dataType[arraySize];`

9.3 Initialization

- `int[] numbers = {1, 2, 3, 4, 5};` (combined declaration, instantiation, and initialization)

9.4 Accessing Elements

Use an index (starting from 0) within square brackets (e.g., `numbers[0]` accesses the first element).

9.5 length Property

`arrayName.length` gives the number of elements in the array.

9.6 Multi-dimensional Array

Tables with rows and columns.

10 Variable Scope

10.1 Definition

The region of a program where a variable can be accessed.

10.2 Rules

- Variables declared inside a method are local to that method.
- Variables declared inside a block (e.g., within an `if` statement or a loop) are local to that block.

11 Comments and Commands

11.1 Comments

Documentation, available in three forms:

- `//` (single-line comment)
- `/* */` (multi-line comment)
- `/** */` (Javadoc/dev comment)

11.2 Command

- Ends with `;`, can span one line or multiple lines.

Chapter 3: Abstraction & Encapsulation

1 Four Principles of Object-Oriented Programming

- **Abstraction**: Focusing on the essential characteristics of an object while hiding unnecessary details. It's about simplifying complex reality.
- **Inheritance**: Creating new classes (derived classes) from existing classes (base classes), inheriting their properties and behaviors. Inheritance as primarily a tool for texttt reuse (not just hierarchical relationships).
- **Encapsulation**: Bundling data (attributes) and methods (operations) that operate on that data within a class. It also involves controlling access to the data (data hiding).
- **Polymorphism**: The ability of objects of different classes to respond to the same method call in their own way. Polymorphism's role in enabling dynamic runtime behavior, substitutability, flexibility and extensibility

2 Abstraction

Definition: Abstraction is the process of focusing on essential characteristics of an entity while hiding unnecessary or complex underlying details. It's about reducing complexity and highlighting the important aspects relevant to a specific context. A more formal definition is: *removing distinctions to emphasize commonalities.*

- **In OOP**: Abstraction allows us to represent real-world objects with simplified models (classes) by focusing only on the relevant properties and behaviors for a particular problem. This manages complexity and makes systems easier to understand and work with.
- **Class as Abstraction**: A class represents an abstraction of a set of objects that share common characteristics (attributes) and behaviors (methods).
- **Levels of Abstraction**: Abstraction can exist at multiple levels. You can have very general, high-level abstractions and more specific, detailed abstractions.

- **Abstraction and Context**: The specific view of an entity, including only related properties, depends on the context in which the entity is being considered.
- **Benefits**: Abstraction simplifies complex systems, makes technology easier to use, and promotes maintainability.

Example: When modeling a washing machine, we might abstract away the low-level electronic details and focus on operations like `startCycle`, `setTemperature`, etc.

3 Class Building

3.1 Class vs. Object

- **Class**: A blueprint or template for creating objects. It defines the attributes (data) and methods (behavior) that objects of that class will have. A class is the result of the abstraction process.
- **Object**: An instance of a class. A concrete realization of the class blueprint, with its own unique set of data values (state).
- **Method**: The implementation of behavior/operation.

3.2 Class Representation in UML

A class is represented in UML by a rectangle with three compartments:

- **Top**: Class name.
- **Middle**: Attributes (data members).
- **Bottom**: Operations/Behavior (member functions).

3.2.1 Visibility

Controls access to class members (both attributes and methods).

- **Public (+)**: Accessible from anywhere.
- **Private (-)**: Accessible only within the class itself (enforces data hiding).
- **Protected (#)**: Accessible within the class and its subclasses.
- **Package/Default (∼ or no symbol)**: Accessible within the same package.

3.3 Class Construction

3.3.1 Class Members

- **Attributes (Data Elements/Fields)**: Named characteristics of a class. All instances have these attributes, but with potentially different values.
- **Operations/Methods**: The actions or behaviors an object can perform. Methods are the implementations of these operations.

3.3.2 Method Signature

- Uniquely identifies a method. Consists of the method name and the number and types of its parameters.
- **Return Type**: The type of data a method returns (e.g., `int`, `String`, `void` if it returns nothing).

3.4 Class Declaration

```java
package packagename;

access_modifier class ClassName {
    // access_modifier: public, or none
    // Inside class: Declare attributes/methods
}
```

4 Encapsulation and Data Hiding

- **Encapsulation**: Encapsulation is the bundling of data (attributes) and methods (operations) that operate on that data within a class. It's the mechanism that achieves data hiding, other objects must access private data via public functions.
- **Data Hiding**: The practice of making attributes private to restrict direct access from outside the class. This protects data integrity and allows for internal changes without affecting external texttt, private data accessed only via public methods. This is achieved by using access modifiers.
- **Two Views of an Object**:
  - Internal view: Details on attributes and methods of the corresponding class.
  - Client (External) view: Services provided by the object and how the object communicates with all the rest of the system.

4.1 Access Modifiers

Include table for `public`, `private` and default access across same class, same package, and different packages.
Keywords that control the visibility of class members (attributes and methods):

- `public`: Accessible from anywhere, if access on other class on differences package using `import package.ClassName` or using fullname `package.ClassName`.
- `private`: Accessible only from within the class itself.
- `protected`: Accessible from within the class, subclasses, and other classes in the same package.
- `(default/package)`: No access modifier specified. Accessible from within the same package.

4.2 Benefits of Encapsulation and Data Hiding

- **Data Protection**: Prevents accidental or malicious modification of an object's internal state from outside the class.
- **Maintainability**: Changes to the internal implementation of a class don't affect other parts of the system as long as the public interface remains the same.
- **Flexibility**: Allows for changes to the internal representation of data without breaking texttt that uses the class.
- **Modularity**: Promotes the creation of self-contained, reusable components.

4.3 Accessor and Mutator Methods (Getters and Setters)

- **Accessor (Getter)**: A public method that provides read-only access to a private attribute, must not modify object state (e.g., getBalance() returns value without altering it). Typically named `getX`, where `X` is the attribute name.
- **Mutator (Setter)**: A public method that allows controlled modification of a private attribute. Typically named `setX`, where `X` is the attribute name. Setters often include validation logic to ensure data integrity.

4.4 Mechanism

- Data hiding is achieved by using the `private` access modifier for attributes. Access to the data is then controlled through public get (accessor) and set (mutator) methods.

4.5 Constant Member

- Members whose values cannot be changed after initialization. Declare by using the `final` keyword.

5 Object Creation and Communication

5.1 Package

- **Definition**: A group of classes considered as a directory, a place to organize classes in order to locate them easily.
  - Avoid naming conflicts, organize classes, protect class and their members from outside access and provides access control.
  - Naming conventions: Lowercase letters, separated by dots (e.g., com.megabank.models).
  - Default package behavior (no explicit package declaration), similar to directories/folders for organizing files.
  - In other languages, known as a *namespace*.
- **Syntax**:

```
        package package_name;
```

5.2 Object Declaration and Initialization

- **Object Properties**
  - Identity: The object reference or variable name
  - State: The current value of all fields
  - Behavior: Methods
- **Constructor Use and Definition**: Constructor does not have return value, but when being used with the keyword new, it returns a reference pointing to the new object.
- **Declaration**: Declaring an object variable is like declaring a variable of any other type, but it's crucial to understand that this only creates a reference, not the object itself. The reference is like a pointer, but Java manages memory automatically, so you don't directly manipulate memory addresses.

```
        ClassName objectName;
```

  At this point, objectName holds a special value called null, meaning it doesn't point to any object yet.
- **Instantiation (Creation)**: This is where the actual object is created in memory. The new keyword is used, followed by a call to a constructor of the class.

```
        objectName = new ClassName(arguments); // Calls a constructor of the class
```

  The new operator does two things:
  - Allocates memory on the heap (a region of memory for dynamic allocation) for the new object.
  - Calls the specified constructor of the class to initialize the object's attributes.

- **Combined Declaration and Initialization**: You can combine the declaration and instantiation into a single line:

```java
        ClassName objectName = new ClassName(arguments);
```

- **Array of Objects**: Similar to primitive arrays, but each element is a reference to an object (initially null). You must create each object individually.

```java
        MyClass[] objectArray = new MyClass[10]; // Creates an array of 10 references
        for (int i = 0; i < objectArray.length; i++)
        {
            objectArray[i] = new MyClass(); // Creates an object for each element
        }
```

5.3 Constructor

- **Definition**: Special methods automatically called when an object is created (using the `new` keyword). Used to initialize the object's attributes.
- **Purpose**: To initialize the attributes of an object when it is created. Constructors ensure that objects start in a valid and consistent state.

```java
        public ClassName()
        {
            // some statement
        }

        public static void main(String args[])
        {
            ClassName object = new ClassName();
        }
```

- **Characteristics**:
  - Has the same name as the class.
  - Can not be considered as a class member.
  - Does not have a return type (not even `void`).
  - Can have parameters (to provide initial values for attributes).
  - A class can have multiple constructors (overloading) but at least one constructor.
  - If you don't define any constructor, Java provides a default, no-argument constructor. If you define any constructor, Java does not provide a default constructor.
  - Constructor access modifiers (public, private, etc.) and restrictions (cannot use keywords like abstract, static, final, native, or synchronized).
- **Types of Constructors**:
  - **Default Constructor**:
    - A constructor with no parameters.
    - If you don't define any constructors in your class, Java automatically provides a default constructor that does nothing (it initializes numeric fields to 0, booleans to false, and object references to null).
    - If you do define any constructor (even one with parameters), Java does not provide a default constructor. This is a common source of errors. If you want a no-argument constructor, you must explicitly define it.
    - JVM provides a default constructor if none is defined, with the same access level as the class.

  - **Parameterized Constructor**: A constructor that takes one or more parameters. These parameters are used to provide initial values for the object's attributes.
  - **Constructor Access Modifiers**: Like other methods, constructors can use access modifiers.
  - **Constructor Note**: Constructors are not inherited.

5.4 Object Usage

- **Accessing Members**: Use the dot (.) operator to access an object's public members (attributes and methods).

```
        objectName.attributeName
        objectName.methodName(arguments)
```

- **Self-Reference(`this`)**: Refers to the current object within a class. Used to access the object's own members and resolve naming conflicts.
  - Allows to access to the current object of class.
  - Is important when function/method is operating on two or many objects.
  - Removes the name conflict between a local variable, parameters and data attributes of class.
  - Is not used in static texttt block

5.5 Object Lifecycle

- **Creation**: Memory is allocated, and the constructor is called.
- **Destruction**: Memory is released (handled automatically by garbage collection in Java; requires explicit deallocation in languages like C++).

Example:

```java
public class BankAccount
{
    private String owner;
    private double balance;
    public BankAccount()
    {
        // some statement
    }
    public void setOwner(String owner)
    {
        this.owner = owner;
    }
    public String getOwner()
    {
        return owner;
    }
}

public class Test
{
    public static void main(String args[])
    {
        BankAccount acc1 = new BankAccount();
        BankAccount acc2 = new BankAccount();
        acc1.setOwner(\Hoa");
        acc2.setOwner(\Hong");
        System.out.println(acc1.getOwner() + \ " + acc2.getOwner());
    }
}
```

Chapter 4: Some Techniques in Class Building

1 Method Overloading

- **Definition:** Methods within the same class can have the same name but must have different signatures. This means either a different number of arguments, or if the number of arguments is the same, the types of the arguments must differ.
- **Purpose:** Allows you to perform the same basic task (represented by the method name) on different data types or with different numbers of inputs. Improves texttt readability and reduces the need to remember many different method names.
- **Example:** The `System.out.println()` method is heavily overloaded, accepting various data types (`int`, `double`, `String`, `boolean`, `char[]`, etc.).
- **Rules:**
  - Overloaded methods must be in the same class.
  - The compiler uses the number and/or types of arguments to determine which overloaded method to call.
  - Return types are not considered part of the signature for overloading purposes. You can't have two methods with the same name and parameters but different return types.
  - It is applied to methods that describe the same kind of task.

2 Constructor Overloading

- **Definition:** Similar to method overloading, a class can have multiple constructors, each with a different set of parameters.
- **Purpose:** Provides flexibility in how objects of the class are created. You can have a default constructor (no arguments), constructors with some parameters, and constructors with all parameters.
- **Example:** A `BankAccount` class might have a constructor with no arguments (setting default values), a constructor with an account number, and a constructor with an account number and an initial balance.
- **Rules:**
  - Follows the same principles of method overloading.
  - Any number of constructors with different parameters.

3 The `this` Keyword

- **Definition:** A reference to the current object within an instance method or constructor.
- **Purpose:**
  - **Distinguish instance variables from local variables:** If a method parameter or local variable has the same name as an instance variable, `this.variableName` refers to the instance variable, while `variableName` refers to the local variable or parameter.
  - **Call another constructor:** Within a constructor, `this(...)` can be used to call another constructor of the same class. This must be the first statement in the constructor.
- **Examples:**
  - `this.owner = owner;` (In a `BankAccount` class, distinguishes the instance variable `owner` from a parameter also named `owner`.)
  - Constructor chaining:

```java
    public BankAccount()
    {
        this("No Name");
    }
```

(A no-argument constructor calls a constructor that takes a name.)

4 Constant Members

- **Definition:** Attributes (fields) that cannot change their value after initialization.
- **Declaration:** Use the `final` keyword. Often combined with `static`.
- **Syntax:** `access_modifier final data_type CONSTANT_NAME = value;`
- **Convention:** Constant names are typically written in all uppercase with underscores separating words (e.g., `MAX_VALUE`, `SECONDS_PER_YEAR`).
- **Purpose:** Define values that are fixed and should not be modified, improving texttt clarity and preventing accidental changes.

5 Class Members (Static Members)

- **Definition:** Attributes (fields) and methods that belong to the class itself, rather than to individual instances (objects) of the class.
- **Declaration:** Use the `static` keyword.
- **Purpose:**
  - **Shared data:** Static fields are shared by all instances of the class. A change to a static field in one object is reflected in all other objects of that class.
  - **Utility methods:** Static methods often perform operations that don't depend on the state of a particular object. They are often used for utility functions related to the class.
  - Accessing without instance.
- **Example:** `Math.sqrt(x)` (The `sqrt` method is a static method of the `Math` class; you don't need a `Math` object to use it).
- **Rules:**
  - Static methods can only access other static members (fields and methods) of the same class. They cannot directly access instance variables or call instance methods (without an object reference).
  - Instance methods can access both static and instance members.

6 Passing Arguments to Methods

- **Java is Pass-by-Value:** Java always uses pass-by-value. This means that a copy of the argument's value is passed to the method.
- **Primitive Types:** When you pass a primitive type (`int`, `double`, `boolean`, etc.), a copy of the value is passed. Changes made to the parameter within the method do not affect the original variable outside the method.
- **Reference Types (Objects and Arrays):** When you pass an object or an array, a copy of the reference (the memory address) is passed. This means:
  - You cannot change the original reference itself (make it point to a different object).
  - You can modify the contents of the object or array that the reference points to. These changes will be visible outside the method because both the original reference and the copied reference point to the same object in memory.

7 Varargs (Variable Arguments)

- **Definition:** Allows a method to accept a variable number of arguments of the same type.
- **Syntax:** Use an ellipsis (`...`) after the data type in the parameter list:

```java
    public void myMethod(int... numbers)
    {
        //Some statement
    }
```

- **Usage:** Inside the method, the varargs parameter is treated as an array. You can call the method with zero or more arguments of the specified type, or you can pass an array directly.
- **Example:**

```java
    public void printValues(String... values)
    {
        //Some statement
    }
```

Can be called as `printValues("a", "b", "c")` or `printValues(new String[]{"a", "b", "c"})`.

Chapter 5: Memory Management & Class Organization

1 Memory Management in Java

- **No Pointers (Directly):** Java manages memory addresses automatically. You don't work with pointers directly, reducing risks of accidental memory corruption.
- **JVM Control:** The Java Virtual Machine (JVM) handles memory allocation and deallocation transparently.

1.1 Heap Memory

- Where objects created with the `new` keyword reside.
- Reference variables (like `String s`) stored on the stack point to objects on the heap.
- Multiple references can point to the same object.

1.2 Stack Memory

- Stores local variables (method parameters, variables declared within methods).
- Primitive type variables (`int`, `float`, `boolean`, etc.) store their actual values directly on the stack.
- Reference type variables store the memory address (reference) of the object (which lives on the Heap).
- **Example (Heap vs. Stack):**

```java
public class MemoryExample
{
    public static void main(String[] args)
    {
        int primitiveOnStack = 10; // Value 10 stored directly on stack frame for main()
        String referenceOnStack = new String("Heap Object"); // referenceOnStack (address) on stack,
        // the String object "Heap Object" is on the Heap

        DataObject dataRef = new DataObject(5); // dataRef (address) on stack,
        // the DataObject instance is on the Heap

        methodCall(dataRef);
        System.out.println(dataRef.getValue()); // Output: 100(original object on heap was modified)
    }

    public static void methodCall(DataObject objParam)
    {   // objParam (address copy) on stack for methodCall()
        // objParam points to the SAME object on the Heap as dataRef in main()
        objParam.setValue(100);
    }
}

class DataObject
{
   private int value;
   public DataObject(int v) { this.value = v; }
   public void setValue(int v) { this.value = v; }
   public int getValue() { return value; }
}
```

1.3 Garbage Collector (GC)

- An automatic background process in the JVM.
- Periodically identifies and reclaims memory used by objects that are no longer referenced (no active variable points to them).
- Makes manual memory deallocation unnecessary (unlike C/C++).
- `System.gc()`: Suggests running the GC, but doesn't guarantee immediate execution.
- **Example (GC Eligibility):**

```java
public class GCDemo
{
    public static void main(String[] args)
    {
        String importantData = new String("Keep this");
        String tempData = new String("Use and discard");

        System.out.println(tempData); // Use the object

        // Make tempData object eligible for GC:
        tempData = null; // The reference no longer points to the object.

        if (true)
        {
            String scopedData = new String("Only lives here");
            System.out.println(scopedData);
        } // scopedData reference goes out of scope here.
          // The "Only lives here" object is now eligible for GC.

        System.out.println("Requesting GC (timing is not guaranteed)");
        System.gc(); // This is just a suggestion to the JVM.

        System.out.println(importantData); // Still accessible
    }
}
```

- `finalize()`: A method inherited from `Object`. The GC might call this method on an object just before reclaiming its memory. It's **NOT** a reliable destructor like in C++. Its use is discouraged for most resource cleanup (files, sockets should be closed explicitly) due to unpredictable timing.
- **Example (`finalize()` - use with caution):**

```java
class ResourceNeedingCleanup
{
    private String name;
    public ResourceNeedingCleanup(String name)
    {
        this.name = name;
    }

    @Override
    protected void finalize() throws Throwable
    {
        try
        {
            // WARNING: Not guaranteed to run! Better to use try-with-resources or finally blocks.
            System.out.println("finalize() called for " + name + ". Performing non-critical cleanup.");
        }

        finally
        {
            super.finalize(); // Always call super.finalize()
        }
    }
}
```

1.4 Object Comparison

- `==` (Primitives): Checks if the values are identical.
- `==` (Objects): Checks if two reference variables point to the exact same object in memory (identity comparison).
- `.equals()` (Objects): Inherited from `Object` (defaults to `==`). Intended for logical equality comparison (do the objects represent the same value/state?). Classes like `String` and Wrapper classes override `equals()` for meaningful value comparison. You should override `equals()` in your own classes if you need to compare their state, not just their memory location.
- **Example (`==` vs `.equals()`):**

```java
String literal1 = "Test"; // From string pool
String literal2 = "Test"; // Likely points to the same pool object as literal1
String obj1 = new String("Test"); // Guaranteed new object on heap
String obj2 = new String("Test"); // Another new object on heap

System.out.println("literal1 == literal2: " + (literal1 == literal2)); // Likely true
System.out.println("literal1 == obj1: " + (literal1 == obj1));         // false
System.out.println("obj1 == obj2: " + (obj1 == obj2));                 // false

System.out.println("literal1.equals(literal2): " + literal1.equals(literal2)); // true
System.out.println("literal1.equals(obj1): " + literal1.equals(obj1));         // true
System.out.println("obj1.equals(obj2): " + obj1.equals(obj2));                 // true
```

2 Class Organization

2.1 Packages

- Java's mechanism for organizing related classes and interfaces into namespaces (like folders).
- **Purpose:** Prevent naming conflicts, control access, improve maintainability.
- Packages can contain other packages.
- Standard naming convention uses reverse domain names (e.g., `com.megabank.models`).
- **Fully Qualified Class Name:** The class name prefixed with its full package name (e.g., `java.lang.String`, `com.megabank.models.BankAccount`).

2.2 `import` Statement

- Allows you to refer to classes from other packages using their simple names instead of their fully qualified names.
- You can import a specific class (`import java.util.ArrayList;`) or all classes in a package (`import java.util.*;`).
- **Example (Packages and Import):**

```java
// File: com/example/utils/StringUtils.java
package com.example.utils;
public class StringUtils
{
    public static boolean isEmpty(String s)
    {
        return s == null || s.length() == 0;
    }
}

// File: com/example/app/Main.java
package com.example.app;
import com.example.utils.StringUtils; // Import the specific class

public class Main
{
    public static void main(String[] args)
    {
        String name = "";
        // Use the imported class with its simple name
        if (StringUtils.isEmpty(name))
        {
            System.out.println("Name is empty.");
        }
        // Or use the fully qualified name (no import needed)
        // if (com.example.utils.StringUtils.isEmpty(name))
        // {
        // ...
        // }
    }
}
```

2.3 Basic Java Packages

Java provides a rich set of standard libraries organized into packages:

- `java.lang`: Fundamental classes (`Object`, `String`, `System`, `Math`, Wrapper classes). Automatically imported into every Java program.

- java.util: Utility classes (Collections framework, Date, Scanner).
- java.io: Input/Output operations (files, streams).
- javax.swing: GUI components (older framework).
- Others like java.net, java.sql, java.math, etc.

3 Utility Classes (Examples)

3.1 Wrapper Classes (in java.lang)

- Provide object representations for primitive types (e.g., Integer for int, Double for double, Boolean for boolean).
- Allow primitives to be treated as Objects (e.g., used in Collections).
- Provide useful static utility methods (e.g., Integer.parseInt(), Double.valueOf()) and constants (e.g., Integer.MAX_VALUE).
- **Example (Wrapper):**

```java
int primInt = 42;
Integer wrapInt = primInt; // Autoboxing: primitive -> Wrapper object
int primAgain = wrapInt;   // Autounboxing: Wrapper object -> primitive

String numberStr = "123";
int parsed = Integer.parseInt(numberStr); // Static utility method

System.out.println("Wrapper value: " + wrapInt);
System.out.println("Parsed value: " + parsed);
```

3.2 String Class (in java.lang)

- Represents sequences of characters.
- **Immutable:** Once a String object is created, its value cannot be changed. Operations like concatenation (+) create new String objects.
- Use .equals() for content comparison, not ==.
- String literals ("...") are often pooled by Java for efficiency. new String("...") always creates a new object.
- Provides many useful methods (length(), charAt(), substring(), indexOf(), toLowerCase(), equals(), equalsIgnoreCase(), etc.).
- **Example (String Immutability):**

```java
String base = "Hello";
String combined = base + " World"; // Creates a NEW String object "Hello World"
String upper = base.toUpperCase(); // Creates a NEW String object "HELLO"

System.out.println(base);     // Output: Hello (original is unchanged)
System.out.println(combined); // Output: Hello World
System.out.println(upper);    // Output: HELLO
```

3.3 StringBuffer / StringBuilder (in `java.lang`)

- Represent mutable sequences of characters.
- Use these when you need to modify a string frequently (especially in loops) as they are more efficient than repeated `String` concatenation.
- `StringBuffer`: Thread-safe (slower).
- `StringBuilder`: Not thread-safe (faster). Use this unless multiple threads will modify it concurrently.
- Methods: `append()`, `insert()`, `delete()`, `toString()`.
- **Example (String Immutability):**

```java
StringBuilder builder = new StringBuilder("Initial");
builder.append(" value"); // Modifies the existing builder object
builder.insert(0, "My "); // Modifies again
builder.append("!");

String finalString = builder.toString(); // Convert back to String when done
System.out.println(finalString); // Output: My Initial value!
```

3.4 Math Class (in `java.lang`)

- Provides common mathematical functions (`sqrt`, `pow`, `sin`, `cos`, `random`, `abs`, `max`, `min`, etc.) and constants (`PI`, `E`).
- All methods and constants are static, so you call them directly on the class: `Math.sqrt(x)`, `Math.PI`.
- **Example (Math):**

```java
double radius = 5.0;
double area = Math.PI * Math.pow(radius, 2); // Using static constants and methods
double sqrtVal = Math.sqrt(144.0);

System.out.println("Area: " + area);
System.out.println("Square root: " + sqrtVal);
```

Chapter 6: Aggregation and Inheritance

1 Source code Re-usability

- **Goal:** Reuse existing, tested code instead of rewriting it.
- **Motivations:**
  - Reduce development time/cost
  - Improve software quality and maintainability
  - Better model the real world
- **OOP Approaches Discussed:**
  - **Aggregation:** Reusing code by composing objects (using objects of existing classes within a new class).
  - **Inheritance:** Reusing code by creating specialized classes based on existing ones (extending classes).

2 Aggregation (“Has-A” Relationship)

- **Concept:**
  - Represents a “whole-part” relationship. A “whole” class is composed of, or has, objects of other “part” classes as its members (e.g., a Car has-a Door, a Quadrangle is-part-of 4 Points).
  - Building a complex object (the “whole” or “aggregate”) by composing it from other, existing objects (the “parts”). The whole class has member variables that are objects of the part classes.
- **Terminology:**
  - **Aggregate/Whole Class:** The class that contains objects of other classes (e.g., Quadrangle, Game).
  - **Member/Part Class:** The class whose objects are contained within the aggregate (e.g., Point, Player, Die).
- **Re-use Mechanism:** Re-uses functionality by delegating tasks to its member objects. The aggregate class uses the features of the part objects it contains.
- **UML Representation:**
  - An association line connecting the whole and part classes.
  - An open diamond symbol attached to the “whole” class end of the line.
  - **Multiplicity:** Numbers or symbols (like * for many, 1, 0..1, 4) at each end indicate how many instances of one class relate to an instance of the other (e.g., 1 Quadrangle has 4 Points).
- **Initialization:**
  - Member objects (“parts”) must be initialized before the aggregate (“whole”) object is fully constructed. This typically happens within the aggregate class’s constructor.
  - When an aggregate object is created, its member objects must also be created/initialized. Typically, the aggregate class’s constructor is responsible for initializing its member objects. (“Member objects must be initialized first”).
- **Example:** A Game class might contain Player objects, Die objects, and an Arbitrator object as members.

```java
// Part Classes
class Engine
{
    public void start()
    {
        System.out.println("Engine Started.");
    }
}
class Wheel
{
    //...
}

// Whole Class
class Car
{
    private Engine carEngine;  // Car 'has-an' Engine
    private Wheel[] carWheels; // Car 'has' Wheels

    public Car()
    {
        // Initialize the parts when the Car is created
        this.carEngine = new Engine();
        this.carWheels = new Wheel[4];
        for (int i = 0; i < 4; i++)
        {
            this.carWheels[i] = new Wheel();
        }
        System.out.println("Car created with Engine and Wheels.");
    }

    // Delegate behavior to the part
    public void start()
    {
        System.out.println("Starting car...");
        this.carEngine.start(); // Call the Engine's start method
    }
}

// --- Demo ---
Car myCar = new Car();
myCar.start(); // Output involves Engine's behavior
```

3 Inheritance ("Is-A" Relationship)

- **Concept:**
  - Represents an "is-a-kind-of" relationship. A new class (subclass) is created as a specialized version of an existing class (superclass), inheriting its properties and behaviors.
  - Creating a new class (the subclass, child class, or derived class) based on an existing class (the superclass, parent class, or base class). The subclass automatically acquires (inherits) the non-private properties (attributes) and behaviors (methods) of its superclass.
- **Re-use Mechanism:** Directly re-uses the source texttt (members) defined in the superclass via the class definition itself.
- **Terminology:**
  - **Superclass/Parent/Base Class:** The existing class being inherited from (e.g., Quadrangle, Mammal, Person).
  - **Subclass/Child/Derived Class:** The new class that inherits (e.g., Square, Whale, Employee).
  - **Siblings:** Child classes sharing the same direct parent.
  - **Ancestors:** Parent, grandparent, etc., classes in the hierarchy.

3.1 Subclass Capabilities

- Inherits or Derives non-private members.
- Can add new attributes and methods (Extension).
- Can modify inherited behavior (Method Overriding - changing the implementation of an inherited method).
- **Polymorphism:** An object of a subclass can be treated as an object of its superclass.
- **UML Representation:**
  - A line connecting the child and parent classes.
  - A closed, empty triangle symbol pointing towards the parent (superclass).
- **Hierarchy:** Classes form an "is-a" tree. In Java, `java.lang.Object` is the ultimate root of all classes. Classes with the same parent are "siblings".

3.2 Inheritance Rules & Access Modifiers

- **What is Inherited:** `public` and `protected` members (attributes and methods) are inherited. `default` (package-private) members are inherited only if the subclass is in the same package as the superclass.
- **What is NOT Inherited:** `private` members are not inherited. Constructors and Destructors are not inherited (though superclass constructors are called during subclass object creation).
- `protected` **Access Modifier:** Allows access within the defining class, any subclasses (regardless of package), and any class within the same package. This is key for allowing subclasses controlled access to parent internals.
- **Detailed:**
  - `public`: Inherited and accessible everywhere.
  - `protected`: Inherited and accessible in the subclass and within the same package. Crucial for allowing subclasses access while restricting external access.
  - `default` (package-private): Inherited only if the subclass is in the same package as the superclass.
  - `private`: Members are not directly accessible by the subclass (though they exist within the inherited object structure).
  - **Constructors:** Not inherited, but the subclass constructor must call a superclass constructor.

3.3 Constructor Chaining (Java `super()`)

- The very first statement in a subclass constructor must be a call to a superclass constructor, either implicitly or explicitly.
- **Implicit:** If no `super(...)` call is written, Java inserts `super();` (calling the no-argument constructor of the parent). This fails if the parent doesn't have a no-argument constructor.
- **Explicit:** Use `super(arguments...)` to call a specific parent constructor. This is necessary if the parent only has constructors that require arguments.

3.4 Initialization/Destruction Order

- **Construction:** When a subclass object is created, the superclass constructor is always executed first. This happens either implicitly (if the parent has a no-argument default constructor) or explicitly using `super(...)` as the very first statement in the subclass constructor. The chain goes up to Object.
- **Destruction (finalize):** When an object is garbage collected, `finalize()` methods (if defined) are called in the reverse order of construction (subclass finalize runs before superclass finalize). Note: Relying on finalize is generally discouraged in modern Java.

3.5 Inheritance Syntax in Java

- **Syntax:** `<Subclass> extends <Superclass>`
- **Example:**

```java
// Superclass
class Vehicle
{
    protected String model; // Accessible by Truck

    public Vehicle(String model)
    {
        // Constructor needs to be called by subclass
        System.out.println("Vehicle constructor called.");
        this.model = model;
    }

    public void drive()
    {
        System.out.println("Vehicle " + model + " is driving.");
    }
}

// Subclass
class Truck extends Vehicle
{
    // Truck 'is-a' Vehicle
    private int loadCapacity;

    public Truck(String model, int capacity)
    {
        super(model); // Explicit call to Vehicle(String) constructor - REQUIRED
        System.out.println("Truck constructor called.");
        this.loadCapacity = capacity;
    }

    // New method specific to Truck
    public void haul()
    {
        System.out.println("Truck " + model + " is hauling " + loadCapacity + "kg.");
    }

    // Override method from Vehicle
    @Override
    public void drive()
    {
        System.out.print("Truck specific drive: ");
        super.drive(); // Optionally call the superclass version
    }
}

// --- Demo ---
Truck myTruck = new Truck("BigRig", 5000);
myTruck.drive();      // Calls the overridden drive method
myTruck.haul();       // Calls Truck's own method
System.out.println(myTruck.model); // Accesses protected inherited member
```

4 Aggregation vs. Inheritance

- **Similarity:** Both achieve texttt re-use in OOP.
- **Key Difference:**
  - **Inheritance:**
    - “Is-a-kind-of” relationship.
    - Reuses via class extension.
    - Creates tightly coupled relationship (child depends heavily on parent).
    - Primary mechanism for Polymorphism.
    - Example: Square extends Quadrangle.
  - **Aggregation:**
    - “Has-a” or “is-a-part-of” relationship.
    - Reuses via object composition.
    - Creates looser coupling (whole class interacts with part objects via their public interface).

    * Promotes flexibility.
    * Example: Quadrangle has Point objects.

Chapter 7: Abstract Class and Interface

1 Method Redefinition / Overriding

- **Definition:** A child class (subclass) provides a specific implementation for a method that is already defined in its parent class (superclass).
- **Condition:**
  - Method in subclass must have the **exact same name**.
  - Method in subclass must have the **exact same signature** (parameter list: same number, types, and order).
  - The **return type** must be the same or a **covariant type** (a subtype of the superclass method's return type).
- **Distinction from Overloading:**
  - **Overloading:** Methods within the *same* class have the same name but *different* signatures. Resolved at compile time.
  - **Overriding:** Methods in a *superclass/subclass* relationship have the same name *and* signature. Resolved at runtime (dynamic binding).
- **Effect:** The overridden method in the subclass *replaces* or *enhances* the inherited implementation. Calls on subclass objects execute the subclass's version.
- **Purpose:** Allows a subclass to provide a specialized behavior for an inherited method, replacing or extending the parent's implementation. Objects of the child class will use the overridden version.
- **Keywords:**
  - `this`: Refers to the *current object instance.*
    * `this.member`: Accesses member of the current object.
    * `this()`: Calls another constructor in the *same* class (must be first statement).
  - `super`: Refers to the *direct parent class* members.
    * `super.member`: Accesses member of the *parent* class (e.g., `super.getDetail()`). Essential for reusing parent logic in overriding methods.
    * `super()`: Calls a constructor of the *parent* class (must be first statement in subclass constructor).
- **Overriding Rules:**
  - Access level cannot be *more restrictive* than the overridden method.
  - **Signature Match:** The overriding method must have the same name and parameter list as the overridden method.
  - **Return Type:** The return type must be the same or a covariant type (a subtype of the parent method's return type). The slides mention "same return data types", which is the stricter, original rule.
  - **Access Modifiers:** The overriding method cannot have a more restrictive access modifier than the overridden method (e.g., if the parent method is protected, the child method can be protected or public, but not private or default).
  - Hierarchy: `public` ¿ `protected` ¿ `default` (package-private) ¿ `private`.
  - Example: If parent method is `protected`, child method can be `protected` or `public`.
- **Cannot Override:**

  - `final` **methods:** Methods marked final in the superclass cannot be overridden.
  - `static` **methods:** Static methods belong to the class, not the instance, and cannot be overridden (they can be hidden).
  - `private` **methods:** Private methods are not inherited and thus cannot be overridden. A subclass can define a method with the same name, but it's unrelated to the parent's private method.
  - Constructors: Not inherited, invoked via `super()`.

- **The `final` Keyword:**
  - `final` **method:** Cannot be *overridden* by subclasses (e.g., `public final String baseName()`). Used for fixed implementations (correctness, efficiency).
  - `final` **class:** Cannot be *subclassed* (e.g., `public final class A`). Used for complete implementations.
  - `final` **variable:** Creates a constant.

2 Abstract Classes

- **Definition:** A class declared using the `abstract` keyword. It serves as a template or base for other classes.
- **Instantiation:** Abstract classes cannot be instantiated directly using `new`.
- **Purpose:**
  - Base class for subclasses.
  - Define a common template (shared state/behavior, required behavior).
  - Enforce structure.
  - Factor out common features.
- **Contents:**
  - **Abstract Methods:** Methods declared with the `abstract` keyword and no implementation (just the signature). They act as placeholders that must be implemented by concrete (non-abstract) subclasses.
  - **Concrete Methods:** Regular methods with implementations.
  - Instance variables, constructors, static methods, etc.
- **Rules:**
  - If a class contains even one abstract method, the class itself must be declared abstract.
  - A subclass inheriting from an abstract class must either:
    * Provide implementations for all inherited abstract methods.
    * Be declared abstract itself.
  - Abstract classes cannot be final.
- **Example Usage:** A **Shape** class with an abstract `calculateArea()` method forces **Circle** and **Square** subclasses to provide their specific implementations.

3 Single and Multiple Inheritance

- **Single Inheritance:** A class inherits from only **one** direct parent class (Java's model for classes).
- **Multiple Inheritance:** A class inherits from **multiple** parent classes (e.g., C++).
- **Java's Approach: No** multiple *class* inheritance due to complexity (e.g., "Diamond Problem"). Uses **Interfaces** for multiple *type* inheritance.

4 Interfaces

- **Definition:** A reference type in Java, similar to a class, primarily containing abstract methods signatures and static final constants (implicitly). It defines a "contract" of what methods a class implementing the interface must provide.
- **Purpose:**
  - Achieve total abstraction.
  - Define a contract or capability that classes can promise to fulfill.
  - Enable a form of multiple inheritance (a class can implement multiple interfaces).
  - Facilitate loose coupling (designing systems where components depend on interfaces rather than concrete implementations).
- **Characteristics (Traditional - Pre-Java 8):**
  - Methods are implicitly `public abstract`.
  - Fields (variables) are implicitly `public static final` (constants).
  - Cannot be instantiated directly.

  - Class uses `implements` keyword.
  - Concrete class must implement **all** interface methods.
  - Syntax: `class MyClass extends Base implements Iface1, Iface2 { ... }`
- **Loose Coupling:** Clients texttt against the interface type, allowing different implementations to be used interchangeably.
- **Java 8+ Enhancements:**
  - `default` **Methods:** Provide concrete implementation within the interface using the `default` keyword. Allows adding methods to interfaces without breaking implementers. Can be overridden. If conflicts arise from multiple interfaces, the implementing class **must** override. Use `InterfaceName.super.methodName()` to call a specific default method.
  - `static` **Methods:** Provide implementation within the interface using `static`. Belong to the interface itself, not inherited by implementers. Called via `InterfaceName.staticMethod()`.
- **Syntax:**

```java
class MyClass extends ParentClass implements Interface1, Interface2
{
    //...
}
```

- **Java 8+ Enhancements:**
  - `default` **methods:** Allow interfaces to provide a default implementation for a method. Implementing classes inherit this default behavior but can override it. Useful for evolving interfaces without breaking existing implementations. If a class implements multiple interfaces with conflicting default methods (same signature), the class must explicitly override that method.
  - `static` **methods:** Allow interfaces to have static methods with implementation. These methods belong to the interface itself and are called using the interface name (e.g., `InterfaceName.staticMethod()`). They are not inherited by implementing classes.

5 Abstract Class vs. Interface

- **Key Differences:**

Feature	Abstract Class	Interface (Pre-Java 8)	Interface (Java 8+)
Methods	Abstract & Concrete	Only Abstract	Abstract, default, static
Variables	Instance & Static (any access, final/non-final)	Only public static final (constants)	Only public static final (constants)
Constructors	Can have constructors	Cannot have constructors	Cannot have constructors
Inheritance	Class extends one abstract class	Class implements multiple interfaces	Class implements multiple interfaces
Access Modifiers	Can use public, protected, private, default	Methods implicitly public	Methods implicitly public (static ok)
Purpose	Define common identity/state/behavior ("is-a")	Define a contract/capability ("can-do")	Define contract/capability, add utility

Table 1: Comparison of Abstract Classes and Interfaces

Chapter 8: Polymorphism and Generic Programming

```java
// Class For Example
class Person
{
    String name = "Person";
    void greet() { System.out.println("Hello from Person!"); }
    static void staticGreet() { System.out.println("Static hello from Person!"); }
    String getDetail() { return "Name: " + name; }
}

class Employee extends Person
{
    double salary = 50000;
    Employee() { name = "Employee"; } // Constructor setting name
    @Override
    void greet() { System.out.println("Hello from Employee!"); }
    static void staticGreet() { System.out.println("Static hello from Employee!"); }
    @Override
    String getDetail() { return super.getDetail() + ", Salary: " + salary; }
    void work() { System.out.println("Employee is working."); }
}

class Manager extends Employee
{
    String department = "Sales";
    Manager() { name = "Manager"; } // Constructor setting name
    @Override
    void greet() { System.out.println("Hello from Manager!"); }
    static void staticGreet() { System.out.println("Static hello from Manager!"); }
    @Override
    String getDetail() { return super.getDetail() + ", Dept: " + department; }
    void manage() { System.out.println("Manager is managing."); }
}
```

1 Upcasting and Downcasting

- **Core Idea:** Handling references and objects within an inheritance hierarchy.
- **Primitive Data Types:**
  - **Widening (Implicit "Upcasting"):** Smaller range to larger range (`byte` → `short`). Automatic.

  - **Narrowing (Explicit "Downcasting"):** Larger range to smaller range (`int` → `short`). Requires explicit cast `(short)`.

- **Object Types (Inheritance):**

  - **Upcasting:** Treating a subclass object as an instance of its superclass.
    - **Mechanism:** Implicit/Automatic.

```java
Employee emp = new Employee();
Person personRef = emp; // Implicit Upcasting: Employee 'is-a' Person

personRef.greet(); // Accesses Person's members (but invokes Employee's override!)
// personRef.work(); // Compile Error! work() not in Person.
System.out.println(personRef.name); // Accesses Person's field
```

    - **Direction:** Moving up the hierarchy.
    - **Effect:** Only superclass members are accessible via the superclass reference. Method calls invoke the *overridden* version if present (dynamic binding).

  - **Downcasting:** Treating a superclass reference (that points to a subclass object) as a subclass instance.
    - **Mechanism:** Explicit cast required.

```java
// personRef from previous example points to an Employee object
Employee empRef = (Employee) personRef; // Explicit Downcasting

empRef.work(); // OK! work() is accessible via Employee reference.
empRef.greet(); // Still invokes Employee's overridden method.

// Safety check before downcasting:
Object obj = new Manager(); // Upcast to Object
if (obj instanceof Employee)
{
    System.out.println("obj is an Employee (or subclass)");
    Employee eCheck = (Employee) obj; // Safe downcast
    eCheck.work();
}

// Risky downcasting:
Person pOnly = new Person();
// Employee riskyEmp = (Employee) pOnly; // Runtime Error! ClassCastException
```

    - **Direction:** Moving down the hierarchy.
    - **Purpose:** Access subclass-specific members.
    - **Risk:** `ClassCastException` at runtime if the object isn't actually an instance of the target subclass. Use `instanceof` for safety checks before casting.

2 Static and Dynamic Bindings

- **Core Idea:** Associating a method *call* with the method *implementation*.

- **Static Binding (Compile-time / Early Binding):**

  - **When:** Decision made at compile time.
  - **Basis:** Declared type of the *reference variable.*
  - **Applies to:** `private`, `final`, `static` methods, constructors, overloaded methods.
  - **Example (Static Methods):** Calls depend on the reference type, not the object type.

```java
Person pRefStat = new Employee();
Employee eRefStat = new Employee();
```

```java

pRefStat.staticGreet(); // Calls Person.staticGreet()
eRefStat.staticGreet(); // Calls Employee.staticGreet()

Person.staticGreet();   // Correct way to call static methods
Employee.staticGreet(); // Correct way to call static methods
```

- **Dynamic Binding (Runtime / Late Binding):**
  - **When:** Decision made at runtime.
  - **Basis:** Actual type of the *object* being referenced.
  - **Applies to:** Instance methods (not `private`, `final`, or `static`). This enables polymorphism.
  - **Example (Instance Methods):** Calls depend on the actual object type.

```java
Person pDyn = new Employee(); // Upcasting
pDyn.greet(); // Calls Employee's overridden greet() method at runtime

Person pMgr = new Manager(); // Upcasting
pMgr.greet(); // Calls Manager's overridden greet() method at runtime
```

3 Polymorphism

- **Definition:** "Many forms". Ability of a reference to refer to different object types, and for the same method call to invoke different implementations based on the actual object type.
- **Key Ingredients:** Inheritance/Interfaces, Method Overriding, Upcasting, Dynamic Binding.
- **Example (Object and Behavior Polymorphism):** Using an array of `Person` references to hold different subtypes.

```java
Person[] people = new Person[3];
people[0] = new Person();
people[1] = new Employee(); // Upcasting
people[2] = new Manager();  // Upcasting

// Loop demonstrating polymorphism
for (Person p : people)
{
    System.out.println("--- Processing Person ---");
    // Object Polymorphism: 'p' refers to Person, Employee, Manager objects
    // Behavior Polymorphism: p.greet() calls the correct override
    p.greet();
    System.out.println(p.getDetail()); // Also polymorphic
    System.out.println("Actual class: " + p.getClass().getSimpleName());

    // To call specific methods, need instanceof and downcasting:
    if (p instanceof Manager)
    {
        ((Manager) p).manage(); // Downcast to call manage()
    }
    else if (p instanceof Employee)
    {
        ((Employee) p).work(); // Downcast to call work()
    }
    System.out.println();
}
```

- **Why Polymorphism?:** Extensibility, Simplicity/Maintainability, Flexibility. Allows adding new subtypes (`Intern`, `Director`) without changing the loop processing `Person` references.

4 Generic Programming

- **Concept:** Writing texttt that works with various types without rewriting for each.
- **Pre-Generics Issues (Using `Object`):** Requires explicit, potentially unsafe downcasting.

```java
java.util.ArrayList objectList = new java.util.ArrayList();
objectList.add("Hello");
objectList.add(123); // Can add different types (potential issue)

// Requires casting and carries risk
String str = (String) objectList.get(0);
// Integer num = (Integer) objectList.get(0); // Runtime ClassCastException!
```

- **Java Generics (Java 1.5+):** Provides compile-time type safety and eliminates explicit casts.

```java
// Type safe list - Compiler enforces only Strings can be added
java.util.List<String> stringList = new java.util.ArrayList<>();
stringList.add("Hello");
stringList.add("World");
// stringList.add(123); // Compile Error! Cannot add Integer to List<String>

// No casting needed, type safety guaranteed
String s1 = stringList.get(0);
for (String s : stringList)
{
    System.out.println(s.toUpperCase());
}

// Generic class example
class Box<T>
{
    private T content;

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

    public T getContent()
    {
        return content;
    }
}

Box<Integer> integerBox = new Box<>();
integerBox.setContent(10);
int num = integerBox.getContent(); // No cast needed
```

- **Java Collections Framework (JCF):** Heavily uses generics (`List<E>`, `Set<E>`, `Map<K, V>`).
- **Wildcards in Generics:** Increase flexibility.
  - `<?>` (Unbounded): Accepts any type. Good for read-only or type-independent ops.

```java
void printList(java.util.List<?> list)
{
    for (Object elem : list)
    { // Can only safely read as Object
        System.out.print(elem + " ");
    }
    System.out.println();
    // list.add(new Object()); // Compile Error! Cannot add to <?>
}
// Can call with List<String>, List<Integer>, List<Employee> etc.
// printList(new java.util.ArrayList<String>(java.util.Arrays.asList("a","b")));
// printList(new java.util.ArrayList<Integer>(java.util.Arrays.asList(1,2)));
```

  - `<? extends Type>` (Upper Bounded): Accepts `Type` or subtypes. Allows reading as `Type`.

```java
// Method to process any list containing Person or its subclasses
void processPersonList(java.util.List<? extends Person> personList)
{
    for (Person p : personList)
    { // Can safely read as Person
        p.greet();
    }
    // personList.add(new Employee()); // Compile Error! Cannot add
}
// Can call with List<Person>, List<Employee>, List<Manager>
// processPersonList(new java.util.ArrayList<Employee>());
```

  - `<? super Type>` (Lower Bounded): Accepts `Type` or supertypes. Allows adding `Type`. Read only as `Object`.

- **Function Call vs. Message Passing**
  - **Function Call (Procedural):** Caller knows exact texttt. Direct invocation.
  - **Message Passing (OOP):** Sender sends request; Receiver object decides implementation (dynamic binding). Decouples request from execution.

Chapter 9: JavaFX

1 Introduction to GUI and JavaFX

1.1 GUI (Graphical User Interface)

A type of user interface that allows users to interact with electronic devices using images and visual indicators rather than text commands.

1.2 Java APIs for GUI

- **AWT (Abstract Windowing Toolkit):** Introduced in JDK 1.0, largely replaced by Swing.
- **Swing:** Enhances AWT, integrated into core Java since JDK 1.2.
- **JavaFX:** A modern framework for building rich client applications.

1.3 JavaFX Features

- **Written in Java:** Available for JVM languages.
- **FXML:** An XML-based declarative markup language for defining user interfaces, separating UI structure from application logic.
- **Scene Builder:** A visual design tool for creating FXML layouts via drag-and-drop.
- **Swing Interoperability:** Allows embedding Swing content in JavaFX applications using `SwingNode`.
- **Built-in UI Controls:** A rich set of controls for developing full-featured applications.
- **CSS-like Styling:** Enables styling applications using CSS.

1.4 History of JavaFX

- Originally by Chris Oliver, acquired by Sun Microsystems.
- JavaFX 8 was an integral part of Java.
- From Java SDK 11 onwards, JavaFX is a separate module.

2 GUI Components in JavaFX (The Scene Graph)

2.1 Stage (`javafx.stage.Stage`)

- The top-level JavaFX container (a window).
- Contains all objects of a JavaFX application.
- Passed as an argument to the `start()` method.
- Has `width` and `height`.
- Displayed using the `show()` method.
- Can have different styles (Decorated, Undecorated, Transparent, etc.) set via `initStyle()`.

2.2 Scene (`javafx.scene.Scene`)

- The container for all content in a JavaFX application.
- Represents the physical contents; contains the scene graph.
- Added to a single Stage.
- Created with a root node and optional dimensions.

2.3 Node (`javafx.scene.Node`)

A visual/graphical object within a scene graph. The base class for all scene graph elements.

- **Types of Nodes:**
  - **Root Node:** The first node in the scene graph.
  - **Parent Node/Branch Node (`javafx.scene.Parent`):** A node that can have children.
    * **Group:** A collective node; transformations apply to all children.
    * **Region:** Base class for UI controls (Chart, Pane, Control).
    * **WebView:** Displays web content.
  - **Leaf Node:** A node without children (e.g., `Rectangle`, `Ellipse`, `ImageView`).

2.4 Scene Graph

A tree-like data structure representing the contents of a scene, with nodes arranged hierarchically.

3 Creating a JavaFX Application

3.1 Application Class

Your main class must extend `javafx.application.Application`.

3.2 `start(Stage primaryStage)` method

- The main entry point for all JavaFX applications. You must implement this abstract method.
- Code for creating the UI (Stage, Scene, Scene graph) goes here.

3.3 `main(String[] args)` method

Launches the JavaFX application by calling `launch(args)`.

3.4 Application Lifecycle Methods

- `init()`: Called after the application instance is created, before `start()`.
- `start()`: Called after `init()`.
- `stop()`: Called when the application is shut down.

3.5 Preparing the Scene Graph

- Create a root node (e.g., `Group`, `StackPane`).
- Add child nodes to the root node: `root.getChildren().add(nodeObject);`

3.6 Preparing the Scene

```
Scene scene = new Scene(rootNode, width, height);
```

3.7 Preparing the Stage

```
primaryStage.setTitle("Title");
primaryStage.setScene(scene);
primaryStage.show();
```

3.8 Basic Drawing/Display

`Line`, `Text` are nodes that can be added to the scene graph.

4 JavaFX

- **UI Controls** (`javafx.scene.control.*`)
  - Core visual elements for user interaction (e.g., `Button`, `Label`, `TextField`, `CheckBox`, `ListView`, `ComboBox`, `ProgressBar`, `Hyperlink`, `Accordion`). Typically placed within Layout Panes.
- **Layout Panes** (`javafx.scene.layout.*`)
  - Containers that manage the arrangement and sizing of their child nodes. `Pane` is the base class.
  - **Common Layout Panes**
    - **HBox:** Arranges nodes in a single horizontal row. (Properties: `alignment`, `spacing`).
    - **VBox:** Arranges nodes in a single vertical column.
    - **Group:** Does not apply any specific layout; children are often at (0,0) and may overlap.
    - **FlowPane:** Wraps nodes in a flow (horizontally or vertically).
    - **GridPane:** Arranges nodes in a grid of rows and columns (useful for forms).
    - **BorderPane:** Arranges nodes in five regions: top, bottom, left, right, and center.
    - **StackPane:** Stacks nodes on top of each other.
    - **AnchorPane:** Allows anchoring nodes to the edges of the pane.

5 Event Handling Models (`javafx.event.*`)

5.1 Event

An action or occurrence recognized by the software (e.g., mouse click, key press). `javafx.event.Event` is the base class.

5.2 Types of Events

- **Foreground Events:** Result from direct user interaction (e.g., `MouseEvent`, `KeyEvent`, `DragEvent`).
- **Background Events:** Do not require direct user interaction (e.g., OS interrupts, timer expiry).

5.3 Event Handling

The mechanism that controls the event and decides what action to take.

5.4 Event Properties

- **Target:** The node on which the event occurred.
- **Source:** The origin of the event.
- **Type:** The specific type of event (e.g., `MOUSE_CLICKED`).

5.5 Event Delivery Process

1. **Target Selection:** The system determines the target node for the event.
2. **Route Construction (Event Dispatch Chain):** A path is built from the Stage down to the target node.
3. **Event Capturing:** The event travels down the dispatch chain from Stage to target. Event *filters* registered on nodes in this path are executed.
4. **Event Bubbling:** After reaching the target, the event travels back up the dispatch chain from target to Stage. Event *handlers* registered on nodes in this path are executed.

5.6 Event Filters and Handlers

- Implement the `EventHandler<T extends Event>` interface.
- **Filter:** Intercepts an event during the capturing phase. Added via `node.addEventFilter()`.
- **Handler:** Processes an event during the bubbling phase. Added via `node.addEventHandler()`.
- An event can be `consume()`d by a filter or handler to stop its further propagation.

6 "Drag and drop" GUI with SceneBuilder (FXML and Controllers)

6.1 Idea

Separate UI definition (FXML) from application logic (Controller).

6.2 Workflow

1. **Install SceneBuilder.**
2. **Design UI in SceneBuilder:**
   - Drag and drop components.
   - Set component properties.
   - Define `fx:id` for components.
   - Define `onAction` (or other event) method names (e.g., `#methodName`).
   - Generates an FXML file.
3. **Create JavaFX Project.**
4. **Copy FXML file** into the project.
5. **Create a Controller Class:**

- Contains logic for the UI.
- Link FXML components using `@FXML` annotation and matching `fx:id`.
- Implement methods for event handlers.
- Optionally implement `Initializable`.

6. **Connect FXML to Controller:**
   - In FXML, set `fx:controller` attribute on the root element (e.g., `fx:controller="com.example.MyControlle`
7. **Load FXML in Application:**
   - In `start()`, use `FXMLLoader`:

```
            Parent root = FXMLLoader.load(getClass().getResource("path/to/your.fxml"));
```

   - Create a `Scene` and set it on the `Stage`.

Chapter 10: Exception Handling

1 Understanding Exceptions

1.1 Definition

An exception is an exceptional event that occurs during the execution of a program, breaking its expected flow. It's a particular type of error or unexpected result.

1.2 Consequence of Unhandled Exceptions

If an exception occurs and is not handled, the program will typically exit immediately, and control will return to the Operating System (OS).

1.3 Classical Error Handling (Limitations)

- Involves writing handling code directly where errors occur, making programs complex.
- Often lacks sufficient information to handle errors effectively.
- Involves sending status codes (flags) via arguments, return values, or global variables, which can be easily misunderstood and hard to maintain.
- **Disadvantages:** Difficult to cover all cases (arithmetic, memory errors), developers often forget to handle errors (due to human factors, lack of experience, or deliberate ignorance).

2 Goals and Models of Exception Handling

2.1 Goals

- Make programs more reliable by avoiding unexpected termination.
- Separate the main logic of the program from the code that handles exceptions. This improves code readability and maintainability.

2.2 Object-Oriented Approach to Exceptions

- Unexpected conditions are packed into an **object**.
- When an exception occurs, an object corresponding to that exception is created, storing detailed information.
- Provides an efficient mechanism for handling errors.
- Separates irregular control threads (exception handling) from regular program flow.

2.3 Handling Location

Exceptions need to be handled either at the method that causes them or be delegated to its caller method.

3 Exception Handling in Java

3.1 Strong Mechanism

Java has a robust, object-oriented mechanism for handling exceptions.

3.2 Throwable Class

All exceptions are representations of a class derived from the `java.lang.Throwable` class or its child classes. These objects carry information about the exception.

3.3 Key Keywords

- `try`: Defines a block of code where exceptions might occur.
- `catch`: Defines a block of code (an exception handler) that handles a specific type of exception caught from the preceding `try` block.
- `finally`: Defines a block of code that is always executed after the `try` block (and any `catch` blocks), regardless of whether an exception occurred or was caught. It's used for cleanup tasks (e.g., closing files).
- `throw`: Used to explicitly throw an exception object.
- `throws`: Used in a method signature to declare the types of exceptions that the method might throw and delegate to its caller.

3.4 `try/catch` Block

- **Purpose:** Separates regular program code from exception handling code.
- **Structure:**

```java
try {
    // Code block that might cause an exception
} catch (ExceptionType1 e1) {
    // Handling for ExceptionType1
} catch (ExceptionType2 e2) {
    // Handling for ExceptionType2
} // ... more catch blocks
```

- **`ExceptionType`:** Must be a descendant of `Throwable`.

3.5 Exception Hierarchy in Java

- **`Throwable`:** The root class of the exception hierarchy.
  - `getMessage()`: Returns a short description of the exception.
  - `printStackTrace()`: Prints the stack trace.
- **`Error` (Subclass of `Throwable`):**
  - Represents serious problems that a reasonable application should not try to catch (e.g., `VirtualMachineError`, `OutOfMemoryError`).
  - Typically unrecoverable. Considered "unchecked exceptions".
- **`Exception` (Subclass of `Throwable`):**
  - Represents conditions that a reasonable application might want to catch.

- **Checked Exceptions:** Subclasses of `Exception` (but not `RuntimeException`) are checked. Compiler forces handling (catch or declare with `throws`). Examples: `IOException`, `SQLException`.
- `RuntimeException` **(Subclass of** `Exception`**):**
  - Represents exceptions during normal JVM operation.
  - These are "unchecked exceptions." Not mandatory to catch. Examples: `NullPointerException`, `ArithmeticException`.

3.6 Nested `try-catch` Blocks

Used when parts of a code block have different error possibilities. The inner `try` block's exceptions are handled first.

3.7 Multiple `catch` Blocks

A `try` block can have multiple `catch` blocks.

- **Order Matters:** Catch more specific (derived) exceptions before more general (base) ones.
- `catch (Throwable e)` catches all.

3.8 `finally` Block

- **Purpose:** Ensures cleanup tasks are performed.
- **Execution:** Always executed after `try` (and any `catch`), regardless of exceptions.
- **Rule:** A `try` block must have at least one `catch` or one `finally` block (or both).

4 Exception Delegation

4.1 Concept

A method can delegate exception handling to its caller.

4.2 `throws` keyword

Used in a method signature to declare checked exceptions it might throw.

```java
public void myMethod() throws IOException, MyCustomException { /* ... */ }
```

4.3 `throw` keyword

Used in a method body to explicitly throw an exception.

```java
if (errorCondition) {
    throw new IOException("File not found");
}
```

4.4 Rules for Delegation

- **Checked exceptions** thrown in a method (or by methods it calls) must be handled (`try-catch`) or declared (`throws`).
- `RuntimeException`**s and** `Error`**s (unchecked)** do not need to be declared with `throws`.

4.5 Caller's Responsibility

If calling a method that `throws` a checked exception, the caller must also handle or delegate it.

4.6 Delegating Multiple Exceptions

A method can declare multiple `throws` types, comma-separated.

4.7 Exception Propagation

If an exception is not handled, it propagates up the call stack (e.g., from `C()` to `B()` to `A()` to `main()`). If unhandled in `main()`, the program terminates.

4.8 Inheritance and Exception Delegation

When overriding a method:

- The overriding method **cannot** throw any new checked exceptions not declared by the parent method.
- Can throw fewer or narrower (subclass) checked exceptions.
- Can throw any `RuntimeException`.

4.9 Advantages of Exception Delegation

- Easy to use.
- Improves readability by centralizing error handling.
- Allows sending control to appropriate handlers.
- Facilitates throwing various exception types.
- Separates error handling from main logic.
- Helps avoid missing exceptions (if declared).
- Allows grouping and categorizing exceptions.

5 User-Defined Exceptions

5.1 Need

Standard exceptions may not cover all application-specific errors.

5.2 Creation

Define custom exception classes by inheriting from `Exception` (for checked) or `RuntimeException` (for unchecked).

- Inherit methods from `Throwable`.
- Provide constructors (e.g., taking a message string, a cause).

```java
public class MyCustomException extends Exception {
    public MyCustomException(String msg) {
        super(msg);
    }
    public MyCustomException(String msg, Throwable cause) {
        super(msg, cause);
    }
}
```

5.3 Usage

Thrown and caught like standard exceptions.

6 Types of Exception Handling (Conclusion)

- Fix errors and call the method again.
- Fix errors and continue running the method (if possible).
- Handle the error differently (alternative logic) instead of ignoring the result.
- Exit the program (as a last resort or if the error is unrecoverable).

Chapter 11: UML, System Modeling

The Unified Modeling Language (UML) is a standardized graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. It provides a set of tools to understand systems from different perspectives.

1 View Model

A complex system cannot be fully understood from a single perspective. The 4+1 View Model addresses this by describing a system from five interlocking viewpoints:

1. **Use-Case View (The "+1"):** Describes the system's functionality from the end-user's perspective. It drives all other views.
   - *Key Diagram:* Use-Case Diagram.
2. **Logical View:** Describes the system's structure, including classes and their relationships. This is the **static view**.
   - *Key Diagrams:* Class Diagram, Object Diagram.
3. **Process View:** Describes the system's concurrency and synchronization. This is a **dynamic view**.
   - *Key Diagrams:* Activity Diagram, Sequence Diagram.
4. **Implementation View:** Describes the organization of source code files and components.
   - *Key Diagram:* Component Diagram.
5. **Deployment View:** Describes the physical deployment of the software onto hardware.
   - *Key Diagram:* Deployment Diagram.

2 Deep Dive into Class Diagrams

Class diagrams are the most fundamental UML diagrams. They model the static structure of a system by showing its classes, attributes, operations, and the relationships among them.

2.1 The Building Blocks: Classes and Objects

- A blueprint or template for creating objects. It defines a common structure (attributes) and behavior (operations) for all objects of its type. For example, a `Professor` class defines that all professor objects will have a `name` and can `teachClass()`.
- An instance of a class. It has its own state (specific values for its attributes) and identity. For example, `ProfessorTorpie` is a specific object of the `Professor` class.

2.2 Anatomy of a Class

A class is represented as a rectangle with up to three compartments:

1. **Name:** The name of the class (e.g., `Student`).
2. **Attributes:** The data or properties of the class (e.g., `name: String`, `studentID: int`).
3. **Operations (Methods):** The behavior or services the class provides (e.g., `getTuition()`, `addSchedule()`).

2.3 Key Class Properties

2.3.1 Visibility

Visibility controls access to attributes and operations, enforcing the principle of encapsulation.

- `+` **Public:** Accessible from any class.
- `-` **Private:** Accessible only within the class itself.
- `#` **Protected:** Accessible within the class and its subclasses (via inheritance).

2.3.2 Scope

Scope determines whether a feature belongs to the class itself or to an instance of the class.

- **Instance Scope (default):** Each object has its own copy of the attribute or operation.
- **Classifier Scope (static):** There is only one copy of the feature, shared by all instances of the class. Denoted by <u>underlining</u> the feature name.

2.4 Organizing Classes: Packages

A **Package** is a mechanism for grouping related UML elements (like classes) into a namespace. It helps manage complexity in large systems, similar to folders in a file system or packages in Java.

3 Class Relationships

Relationships define how classes interact with each other.

3.1 Association

- **What it is:** A general structural relationship indicating that objects of one class are connected to objects of another.
- **Keywords:** ”uses,” ”knows about,” ”is connected to.”
- **Notation:** A solid line between two classes.
- **Key Features:**
  - **Role Names:** Describe the role a class plays in the association (e.g., a `Person` is a *driver* in its association with a `Car`).
  - **Multiplicity:** Defines how many instances of one class can be related to a single instance of another.

Table 2: Common Multiplicity Indicators

Multiplicity	**Meaning**
`1`	Exactly one
`0..1`	Zero or one (optional)
`*` or `0..*`	Zero or more
`1..*`	One or more
`2..4`	A specified range (e.g., 2 to 4)

3.2 Aggregation (Shared "has-a" relationship)

- **What it is:** A special type of association representing a **whole-part** relationship. The part can exist independently of the whole.
- **Notation:** A solid line with an **unfilled (white) diamond** on the "whole" side.
- **Lifecycle:** The part's lifecycle is **not** tied to the whole. If the whole is destroyed, the part may still exist.
- **Example:** A `Car` "has-a" `Door`. The `Door` can exist without the `Car`.

3.3 Composition (Owned "has-a" relationship)

- **What it is:** A stronger form of aggregation where the part is **owned** by the whole and **cannot exist independently**.
- **Notation:** A solid line with a **filled (black) diamond** on the "whole" side.
- **Lifecycle:** The part's lifecycle is **tied** to the whole. If the whole is destroyed, its parts are destroyed too.
- **Example:** A `House` is "composed of" `Rooms`. Destroying the house destroys the rooms.

3.4 Generalization (Inheritance)

- **What it is:** An "is-a-kind-of" relationship where a subclass (child) inherits attributes and operations from a superclass (parent).
- **Notation:** A solid line with a **hollow, triangular arrowhead** pointing to the **superclass**.
- **Example:** A `Savings` account and a `Checking` account are both a kind of `Account`.

4 Overview of Other Key UML Diagrams

- **Use-Case Diagram:** Shows how external **actors** interact with the system's **use cases** (features). Captures functional requirements.
- **Object Diagram:** A snapshot of the system at a particular moment. Shows objects and their links, providing a concrete example of a class diagram's structure.
- **Activity Diagram:** Models the flow of control from one activity to another. Excellent for showing business workflows or complex algorithms.
- **State Machine Diagram:** Describes the different states an object can be in and the events that cause it to transition from one state to another.
- **Interaction Diagrams (e.g., Sequence Diagram):** Shows how a set of objects interact over time to accomplish a task, emphasizing the time-ordering of messages.
- **Deployment Diagram:** Shows the physical hardware nodes and how software components are distributed and deployed across them.