JAVA UNIT 1 NOTES

An Introduction to Java: UNIT - I: Setting the Stage for Advanced Concepts

Before diving into the advanced features of Java, it's helpful to understand the fundamentals of the Java platform itself – its history, key components, how code runs, and its built-in security.

A. Java Versions and Key Features

Java has evolved significantly since its creation by Sun Microsystems (now owned by Oracle). Each major version introduced new features, improving performance, language capabilities, and libraries.

- **Early Versions (JDK 1.0 to 1.4):** Established the core language, Applets (as discussed in Unit 1), AWT, Swing, JavaBeans, JDBC (for databases), RMI (for distributed objects - also in your practical file), and introduced the original `java.io` and the beginnings of `java.net`. JDK 1.4 notably introduced Java NIO.
- **Java 5 (JDK 1.5 / 5.0):** A *major* release. Introduced:
  - **Generics:** Type-safe collections (like `List<String>`) preventing `ClassCastException` at runtime.
  - **Enums:** A fixed set of constants (like days of the week, directions).
  - **Autoboxing/Unboxing:** Automatic conversion between primitive types (like `int`) and their wrapper classes (like `Integer`).
  - **Enhanced `for` loop (for-each loop):** Simplified iteration over arrays and collections.
  - **Variable Arguments (Varargs):** Methods that can accept a variable number of arguments (e.g., `printNumbers(int... numbers)`).
  - **Annotations:** Metadata added to code (`@Override`, `@WebServlet` - seen in your practical file).
- **Java 6 (JDK 6):** Primarily focused on performance improvements, database integration enhancements, and web service support (JAX-WS, JAXB).
- **Java 7 (JDK 7):** Introduced smaller language features (e.g., diamond operator `<>` for simplified generics instantiation, try-with-resources for automatic resource management), NIO.2 (enhancements to the NIO file system API - seen in Unit 1 notes/practical file), and support for dynamic languages on the JVM.
- **Java 8 (JDK 8):** Another *major* release, bringing significant changes for modern programming:
  - **Lambda Expressions:** Concise way to represent anonymous functions (often used with functional interfaces). Key for functional programming style.
  - **Stream API:** Powerful way to process collections of objects (filtering, mapping, reducing) declaratively and efficiently, including support for parallel processing.

    - **Default Methods in Interfaces:** Allowing adding new methods to interfaces without breaking existing implementations.
    - **New Date and Time API:** A much improved API for handling dates, times, durations, etc.
- **Java 9 and beyond (JDK 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21...):** Moved to a faster release cycle (every six months). Key features include:
    - **Modularity (Project Jigsaw - JDK 9):** Organizing the JDK and applications into modules for better encapsulation, reliability, and performance.
    - Improvements to Garbage Collection, JVM performance.
    - New language features (e.g., `var` keyword for local variable type inference - JDK 10, Text Blocks - JDK 15).
    - Pattern Matching enhancements.

Understanding the evolution helps appreciate why certain APIs and concepts exist (like NIO being introduced in 1.4 and enhanced in 7) and how modern Java differs from older versions.

B. JRE vs. JDK vs. JVM

These are three core components of the Java platform, often confused but with distinct roles:

1. **JVM (Java Virtual Machine):**
    - **What it is:** It's an *abstract specification* and a *runtime instance*. It's the engine that executes Java bytecode.
    - **Role:** It reads the `.class` files (which contain bytecode), interprets or Just-In-Time (JIT) compiles the bytecode into native machine code, and runs it. It manages memory (garbage collection) and provides the runtime environment for your Java program.
    - **"Write Once, Run Anywhere":** This is the JVM's magic. The Java code is compiled into *bytecode* (a universal format), and the JVM implementation for a specific operating system and hardware translates and runs that bytecode. This means the same `.class` file can run on Windows, Linux, macOS, etc., as long as a compatible JVM is installed.
2. **JRE (Java Runtime Environment):**
    - **What it is:** It's a *bundle* that contains the JVM and the necessary libraries (Java API classes, also known as the Standard Library or Java Class Library) that your Java programs need to run.
    - **Role:** Provides the minimum requirements to *run* a compiled Java program. If you just want to run existing Java software (like a game or an application), you only need the JRE. It does *not* contain the compiler or development tools.
3. **JDK (Java Development Kit):**
    - **What it is:** It's a *superset* of the JRE. It contains everything in the JRE *plus* development tools like the Java compiler (`javac`), debugger (`jdb`), archiving tool (`jar`), documentation generator (`javadoc`), etc.

    - **Role:** Provides everything you need to *develop, compile, and run* Java applications. As a Java programmer, you primarily work with the JDK.

In simple terms:

- **JVM:** Runs the code (`.class` file).
- **JRE:** JVM + Libraries needed to run the code.
- **JDK:** JRE + Development Tools (like the compiler).

Think of it like this:

- JVM is the car's engine.
- JRE is the engine plus all the necessary parts of the car to make it drive (wheels, transmission, fuel tank).
- JDK is the whole car plus the tools needed to build and repair it (wrenches, screwdrivers, diagnostic equipment).

C. How Java Compilation Works

Java follows a two-step execution process:

1. **Compilation:**
    - You write your Java source code in `.java` files (e.g., `MyProgram.java`).
    - You use the Java compiler (`javac`), which is part of the JDK.
    - The compiler checks your code for syntax errors and translates it into **bytecode**.
    - Bytecode is a low-level, platform-independent instruction set.
    - The compiler saves the bytecode in `.class` files (e.g., `MyProgram.class`).
2. **Execution:**
    - You use the Java Launcher (`java`), which is part of the JRE (and JDK).
    - You tell the launcher the name of the `.class` file you want to run (`java MyProgram`).
    - The launcher starts a **JVM** instance.
    - The JVM loads the necessary `.class` files.
    - The JVM verifies the bytecode for security (see below).
    - The JVM executes the bytecode.
    - Modern JVMs use **Just-In-Time (JIT) compilation**. The JIT compiler identifies frequently executed bytecode and translates it into native machine code for the specific hardware the JVM is running on. This compiled native code runs much faster than interpreted bytecode.

Simplified Flow:

`Your_Code.java (Source Code)` --(javac compiler)--> `Your_Code.class (Bytecode)` --(JVM)--> `Running Program`

This compilation-to-bytecode approach is what makes Java platform-independent.

D. Security Features of Java

Java was designed with security in mind, especially for running code from potentially untrusted sources (like Applets downloaded from the web). Key security features include:

1. **Bytecode Verifier:**
   - When the JVM loads a `.class` file, the Bytecode Verifier examines the bytecode *before* it's executed.
   - It checks if the bytecode is well-formed and adheres to the Java language rules.
   - This prevents malicious code that might have been created by bypassing the standard Java compiler from performing illegal operations (like violating memory access rules).
2. **Security Manager (Sandbox Environment):**
   - Applets and code from untrusted sources typically run within a "sandbox" enforced by the Security Manager.
   - The Security Manager defines a set of permissions (e.g., permission to read/write local files, connect to network resources, access system properties).
   - Code is only allowed to perform actions for which it has explicit permission.
   - This prevents untrusted code from harming the user's system (e.g., deleting files, accessing sensitive data, making unauthorized network connections). (Notes Image 4 mentions Applets are "Secured").
3. **Exception Handling:**
   - As discussed earlier, robust exception handling helps prevent programs from crashing unexpectedly due to runtime errors. This is a form of stability and reliability, contributing to the overall security posture, as crashes can sometimes be exploited.
4. **Automatic Memory Management (Garbage Collection):**
   - Java's automatic garbage collection prevents memory-related errors like memory leaks (where unused memory is not released) and buffer overflows (writing beyond the bounds of a buffer), which are common sources of security vulnerabilities in languages with manual memory management.
5. **Type Safety:**
   - Java is a strongly-typed language, and the compiler and JVM enforce type checking. This prevents errors where code might try to treat data of one type as another, reducing the risk of unexpected behavior or crashes that could be exploited. Generics enhance type safety.

While no system is perfectly secure, these features make Java a relatively secure platform for developing and deploying applications, especially when dealing with code from external sources.

(The rest of the detailed Unit 1 topic explanations remain below, starting with "Advanced Java - Unit I: Deep Dive")

Advanced Java - Unit I: Deep Dive

This unit forms the bridge between fundamental Java programming (Core Java) and building more complex, real-world applications, especially those involving networking, concurrent processing, and web interaction.

1. Introduction to Java / Advanced Java (Core vs. Enterprise)

- **What is it?**
    - Think of Java as having different "editions" tailored for different purposes.
    - **Core Java (J2SE - Java Platform, Standard Edition):** This is what you likely learned first. It's the *foundation*. It provides the basic language features, fundamental data types, object-oriented concepts, basic I/O, collections, etc. It's everything you need to build standard desktop applications and command-line tools. (As seen in Notes Images 1 & 2, called "Cor Java").
    - **Advanced Java / JEE (Java Enterprise Edition):** This is a *set* of specifications and APIs built *on top* of Core Java. Its purpose is to make it easier to build large-scale, distributed, multi-tiered applications that are common in businesses (enterprises). Think of applications that need to handle many users, connect to databases, communicate over networks, and run on servers. (As seen in Notes Images 1 & 2).
- **Why is it important?**
    - Core Java gives you the language basics. Advanced Java gives you the *tools and frameworks* to apply those basics to complex, real-world scenarios like websites, online services, banking systems, etc.
    - It simplifies complex tasks like managing thousands of simultaneous users, ensuring data security across networks, and integrating different software systems. (Notes Image 2 mentions simplifying n-tier applications).
- **Key Concepts & How it Relates to Unit 1:**
    - Advanced Java introduces technologies like Servlets (for web requests), JSPs (for dynamic web pages), EJBs (for business logic), etc. While the unit *syllabus* focuses on fundamental advanced concepts (multithreading, networking), the *practical file* hints at Servlets/JSPs (Programs 17-20). This suggests Unit 1 is laying the groundwork (concurrency, basic networking) needed for those web technologies.
    - The difference between Core (single-tier, desktop focus) and Advanced (multi-tier, web/network focus) highlights *why* you need topics like Socket Programming and URL Connections – they

are crucial for the multi-tier architecture of Advanced/Enterprise applications. (Notes Image 3 table clarifies this).

(No code example needed for this introductory concept difference)

2. Inheritance

- **What is it?**
  - A core Object-Oriented Programming (OOP) concept present in Core Java. It's the mechanism where one class (**subclass** or derived class) inherits properties (attributes) and behaviors (methods) from another class (**superclass** or base class). (Notes Image 1 mentions Inheritance in the syllabus list).
  - Think of it like genetic inheritance – a child inherits traits from their parents.
- **Why is it important?**
  - **Code Reusability:** You don't have to rewrite the same methods or fields in new classes if they share common functionality with an existing class.
  - **Maintainability:** Changes to the superclass are inherited by subclasses, making updates easier.
  - **Polymorphism:** Allows treating objects of different subclasses uniformly through a superclass reference, which is very powerful.
  - **Organization:** Helps structure classes logically, showing relationships between them ("is-a" relationship, e.g., a `Dog` *is-a* `Animal`).
- **Key Concepts:**
  - `extends` keyword: Used to specify that a class is a subclass of another. Example: `class Dog extends Animal { ... }`.
  - Superclass/Subclass: The class being inherited from (superclass) and the class doing the inheriting (subclass).
  - Overriding: Defining a method in the subclass that has the same signature as a method in the superclass.
  - `super` keyword: Used to refer to the superclass's members (constructor, methods, fields).
- **Types of Inheritance in Java (Class Inheritance):**
  - **Single:** One subclass extends one superclass (`Dog extends Animal`).
  - **Multilevel:** A class extends a class, which itself extends another class (`ElectricCar extends Car extends Vehicle`).
  - **Hierarchical:** Multiple subclasses extend the same superclass (`Dog extends Animal`, `Cat extends Animal`).
  - **Hybrid:** A mix of the above. *Direct hybrid inheritance using classes is not supported in Java* (Java doesn't allow a class to extend more than one class). However, you can achieve a form of hybrid inheritance using *interfaces*.
- **Simple Code Example:**

```java
// 1. Superclass
class Animal {
    void eat() {
        System.out.println("Animal eats food.");
    }
}

// 2. Subclass inheriting from Animal (Single Inheritance)
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

// 3. Main class to demonstrate
public class SimpleInheritanceExample {
    public static void main(String[] args) {
        // Create an object of the subclass (Dog)
        Dog myDog = new Dog();

        // Call methods:
        // myDog.eat(); // Calling method inherited from Animal
        // myDog.bark(); // Calling method specific to Dog

        System.out.print("Demonstrating Single Inheritance: ");
        myDog.eat();
        myDog.bark();
    }
}
```

- **Explanation of Code:**
  - We define an `Animal` class with an `eat()` method.
  - We define a `Dog` class that `extends` `Animal`. This means `Dog` automatically gets the `eat()` method from `Animal`.
  - The `Dog` class also adds its own method, `bark()`.
  - In the `main` method, we create a `Dog` object (`myDog`).
  - We can then call both `myDog.eat()` (because it inherited `eat` from `Animal`) and `myDog.bark()` (because it's defined in the `Dog` class).
- **Reference to Practical File:**
  - **Program 1:** This program expands on this basic example to show Multilevel and Hierarchical inheritance with additional classes like `Vehicle`, `Car`, `ElectricCar`, `Employee`, `Manager`, `Developer`.

3. Exception Handling

- **What is it?**
  - A structured way to deal with unexpected events (errors) that occur during the execution of a program. These events, called **exceptions**, disrupt the normal flow of instructions. (Notes Image 1 mentions Exception Handling).
  - Instead of the program crashing, exception handling allows you to "catch" the error and execute specific code to recover or provide a meaningful message.
- **Why is it important?**
  - **Robustness:** Makes your program more reliable and less likely to crash.
  - **Graceful Failure:** Allows the program to shut down cleanly or continue running in a limited capacity, rather than just abruptly terminating.
  - **Separation of Concerns:** Separates the normal program logic from the error-handling logic.
- **Key Concepts:**
  - **Exception:** An event that occurs during the execution of a program that disrupts the normal flow of instructions.
  - **`try` Block:** Contains the code that might potentially throw an exception.
  - **`catch` Block:** Immediately follows a `try` block. Contains the code that is executed if a specific type of exception occurs in the `try` block. You can have multiple `catch` blocks for different exception types.
  - **`finally` Block:** Follows `try` and `catch` blocks. The code in the `finally` block *always* executes, regardless of whether an exception occurred or was caught. Useful for cleanup resources (closing files, network connections).
  - **`throw` Keyword:** Used to explicitly create and throw an exception object.
  - **`throws` Keyword:** Used in a method signature to declare that the method *might* throw a specific type of exception. This informs the caller of the method that they need to handle this potential exception.
- **Types of Exceptions:**
  - **Checked Exceptions:** Exceptions that the compiler *forces* you to handle or declare (`throws`). They typically represent conditions outside the program's immediate control but that a well-written application should anticipate (e.g., `IOException` when reading a file, `SQLException` when dealing with databases, `ClassNotFoundException`).
  - **Unchecked Exceptions (Runtime Exceptions):** Exceptions that occur during program execution and do *not* need to be explicitly handled or declared by the programmer (though you *can* handle them). They often indicate programming errors (e.g., `ArithmeticException` for division by zero, `NullPointerException` for accessing a member of a null object, `ArrayIndexOutOfBoundsException` for accessing an array with an invalid index,

`NumberFormatException` for trying to convert invalid text to a number). (Notes Image 7 mentions examples like Arithmetic, Null Pointer, Array Index Out of Bounds, Number Format).
  - **Errors:** Represent serious problems outside the application's control (e.g., `OutOfMemoryError`, `StackOverflowError`). You typically don't catch these.
  - **Custom Exceptions:** You can create your own exception classes by extending `Exception` (for checked) or `RuntimeException` (for unchecked). (Notes Image 7 shows `CustomException`).
- **Simple Code Example:**

```
import java.io.IOException; // Needed for the checked exception example

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

        // Example 1: Handling an Unchecked Exception
(ArithmeticException)
        System.out.println("--- Unchecked Exception Example ---");
        try {
            int numerator = 10;
            int denominator = 0;
            int result = numerator / denominator; // This line will throw
ArithmeticException
            System.out.println("Result: " + result); // This line will
not be reached
        } catch (ArithmeticException e) {
            // This block is executed if an ArithmeticException occurs
            System.err.println("Error: Division by zero is not
allowed.");
            System.err.println("Exception details: " + e.getMessage());
// Prints "/ by zero"
        } finally {
            // This block always executes
            System.out.println("Arithmetic operation attempt finished.");
        }

        System.out.println(); // Add a blank line

        // Example 2: Declaring and handling a Checked Exception
(IOException)
        System.out.println("--- Checked Exception Example ---");
        try {
            // Call a method that declares it might throw IOException
            simulateFileRead("my_document.txt");
            System.out.println("File read successfully (simulated)."); //
```

```
Won't reach if exception is thrown
        } catch (IOException e) {
            // This block is executed if an IOException occurs
            System.err.println("Error simulating file read.");
            System.err.println("Exception details: " + e.getMessage());
// Prints the exception message
        }
         finally {
            System.out.println("File read attempt finished.");
        }

         System.out.println("\nProgram finished.");
    }

    // A method that declares it might throw an IOException
    static void simulateFileRead(String filename) throws IOException {
        System.out.println("Attempting to simulate reading: " +
filename);
        // In a real scenario, file I/O code would be here.
        // For this example, we just throw an exception to simulate an
error.
        if (filename.contains("non_existent")) { // Simple condition to
throw
             throw new IOException("Simulated: File not found or access
denied.");
        }
         // If the filename doesn't contain "non_existent", it would
proceed here (though we simulate the error)
    }
}
```

- **Explanation of Code:**
    - **Example 1:** The division by zero inside the `try` block causes an `ArithmeticException`. The program flow jumps to the `catch (ArithmeticException e)` block, printing the error message. The `finally` block executes afterward. The `System.out.println("This won't be printed.")` line is skipped because the exception occurred before it.
    - **Example 2:** The `simulateFileRead` method is marked with `throws IOException` in its signature. This tells anyone calling this method that they *must* handle a potential `IOException`. In the `main` method, we call `simulateFileRead` inside a `try` block and provide a `catch (IOException e)` block to handle it. When the simulated exception is thrown, the `catch` block executes. Again, the `finally` block runs afterward.
- **Reference to Practical File:**

- **Program 2:** This program provides clear examples of catching several different *unchecked* exception types within separate `try-catch` blocks and shows the implementation of a custom exception.

4. Multithreading

- **What is it?**
  - Executing multiple parts of a program (called **threads**) concurrently. In a single-core processor, this happens by quickly switching between threads (time-slicing). On multi-core processors, threads can truly run in parallel. (Notes Image 1 mentions Multithreading).
  - A process is a running instance of a program. A thread is a single sequence of execution within a process. A process can have multiple threads running simultaneously.
- **Why is it important?**
  - **Responsiveness:** Prevents your program from "freezing" when performing a long-running task (like fetching data over a network). One thread can handle the task while another keeps the user interface active.
  - **Performance:** Can significantly speed up tasks that can be broken down into smaller, independent parts that run in parallel on multi-core systems.
  - **Handling Multiple Requests:** Essential for servers (like web servers or your custom socket server) to handle multiple client connections at the same time. Each connection can be handled by a separate thread.
- **Key Concepts:**
  - **Thread:** The basic unit of execution.
  - **Creating Threads:**
    - **Extending `java.lang.Thread`:** Create a new class that extends `Thread` and override its `run()` method with the code the thread should execute.
    - **Implementing `java.lang.Runnable`:** Create a class that implements the `Runnable` interface and provides the `run()` method implementation. Then, create a `Thread` object and pass your `Runnable` instance to its constructor (`new Thread(myRunnable)`). This is generally preferred as it allows your class to extend another class if needed.
  - **Starting Threads:** Call the `start()` method of the `Thread` object. This method allocates resources for the new thread and calls the `run()` method. *Calling `run()` directly does not create a new thread; it just runs the `run()` method in the current thread.*
  - **Thread Lifecycle:** Threads go through states like New, Runnable, Running, Blocked/Waiting, Timed Waiting, Terminated.
  - **Synchronization:** A critical concept when multiple threads access **shared resources** (like variables, data structures). Without synchronization, problems like data corruption (race conditions) or deadlocks can occur.

- **`synchronized` keyword:** Can be applied to methods or blocks of code. Ensures that only one thread can execute that method or block at a time for a given object.
- **`wait()`, `notify()`, `notifyAll()`:** Methods of the `Object` class (available to all objects) used for inter-thread communication within `synchronized` blocks. `wait()` causes a thread to release the lock and enter a waiting state until another thread calls `notify()` or `notifyAll()` on the same object.
- **`Lock` Interface (from `java.util.concurrent.locks`):** Offers more flexible and powerful locking mechanisms than the `synchronized` keyword. (Used in Practical Program 7).

- **Simple Code Example (using Runnable):**

```java
// 1. Define the task the thread will perform by implementing Runnable
class MySimpleRunnableTask implements Runnable {
    private String taskName;

    MySimpleRunnableTask(String name) {
        this.taskName = name;
    }

    @Override
    public void run() {
        System.out.println(taskName + " starting. Running in Thread: " +
Thread.currentThread().getName());
        for (int i = 0; i < 3; i++) {
            System.out.println(taskName + ": Step " + i);
            try {
                Thread.sleep(50); // Pause for a short time
            } catch (InterruptedException e) {
                System.out.println(taskName + " was interrupted.");
            }
        }
        System.out.println(taskName + " finished.");
    }
}

// 2. Main class to create and start threads
public class SimpleMultithreadingExample {
    public static void main(String[] args) {
        System.out.println("Main thread starting.");

        // Create instances of the Runnable task
        MySimpleRunnableTask task1 = new MySimpleRunnableTask("Task 1");
        MySimpleRunnableTask task2 = new MySimpleRunnableTask("Task 2");
```

```
        // Create Thread objects, passing the Runnable tasks
        Thread thread1 = new Thread(task1);
        Thread thread2 = new Thread(task2);

        // Start the threads (calls the run() method in each thread)
        thread1.start();
        thread2.start();

        System.out.println("Main thread finished starting other threads.");
        // The main thread continues its execution while thread1 and thread2 run concurrently.
    }
}
```

- **Explanation of Code:**
    - We create a class `MySimpleRunnableTask` that implements the `Runnable` interface. This class contains the `run()` method, which holds the code the thread will execute (printing some messages and pausing).
    - In the `main` method, we create two instances of `MySimpleRunnableTask`.
    - We then create two `Thread` objects, `thread1` and `thread2`, passing the task instances to their constructors.
    - Calling `thread1.start()` and `thread2.start()` tells the Java Virtual Machine (JVM) to create new threads and execute the `run()` method of the associated `Runnable` object within those new threads.
    - The output shows that the tasks are interleaved, demonstrating concurrent execution. The main thread continues executing after starting the other threads.
- **Reference to Practical File:**
    - **Program 5 (Producer-Consumer), Program 6 (Reader-Writer), Program 7 (Dining Philosophers):** These programs provide more complex and realistic examples of multithreading, specifically focusing on concurrency challenges and synchronization mechanisms (`wait`, `notify`, `Lock`). They are excellent examples of how threading is used in advanced scenarios.

5. Applet Programming

- **What is it?**
    - Small Java programs designed to be embedded within HTML web pages and run in a web browser (if the browser has a compatible Java plugin). (Notes Image 4).
- **Why is it important?**

- Historically, applets were one of the first ways to add dynamic, interactive content to web pages before technologies like JavaScript and Flash became dominant. They allowed client-side execution of complex logic.
- **Note:** Applets are now largely deprecated and not supported by modern browsers due to security concerns and the shift to other web technologies. Studying them is often for understanding the evolution of web technologies and client-side Java.

- **Key Concepts:**
  - `java.applet.Applet` class: The base class for all applets.
  - Applet Lifecycle Methods: The browser environment calls these methods in a specific sequence to manage the applet's state. (Notes Images 5 & 6 are very clear on this).
    - `public void init()`: Called once when the applet is first loaded. Use for one-time setup (loading images, initializing variables).
    - `public void start()`: Called after `init()` and whenever the user returns to the page containing the applet. Use for starting threads or resuming activities.
    - `public void paint(Graphics g)`: Called whenever the applet needs to redraw itself (e.g., initially, when resized, when covered and uncovered). The `Graphics` object (`java.awt.Graphics`) is used for drawing shapes, text, and images. (Notes Image 6 explains the Graphics object).
    - `public void stop()`: Called when the user leaves the page. Use for pausing threads or activities.
    - `public void destroy()`: Called once when the applet is unloaded from the browser (e.g., browser window closed). Use for final cleanup (releasing resources).
  - HTML `<applet>` tag: Used in HTML to embed the applet, specifying the `.class` file, width, height, and potentially parameters. (Notes Image 7 shows an example HTML).
  - Applet Viewer (`appletviewer` tool): A command-line tool provided by the JDK to run applets directly from an HTML file (or even a Java file with the `<applet>` tag in comments) without a web browser. Useful for testing. (Notes Image 6, 7, 8).
  - GUI Elements (`java.awt`, `javax.swing`): Applets often use AWT (Abstract Window Toolkit) or Swing for their graphical user interfaces. The `paint` method is from AWT.
  - Applet Security Sandbox: Applets run in a restricted environment to prevent malicious code from accessing local system resources.

- **Simple Code Example (Draws a rectangle and text):**

```java
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color; // Import Color class for drawing colors

/*
This is an HTML comment block that the appletviewer tool can read.
To run this applet using appletviewer:
```

```
1. Save this code as SimpleDrawingApplet.java
2. Compile: javac SimpleDrawingApplet.java
3. Run: appletviewer SimpleDrawingApplet.java

<applet code="SimpleDrawingApplet.class" width="300" height="150">
</applet>
*/

public class SimpleDrawingApplet extends Applet {

    // The paint method is automatically called by the
browser/appletviewer
    // when the applet needs to be drawn.
    @Override
    public void paint(Graphics g) {
        // Set the drawing color to blue
        g.setColor(Color.blue);

        // Draw a filled rectangle (x, y, width, height)
        g.fillRect(20, 20, 100, 50); // Draw at (20, 20) with width 100,
height 50

        // Set the drawing color to red
        g.setColor(Color.red);

        // Draw a string (text, x, y)
        // The y coordinate for text is the baseline of the text
        g.drawString("Hello from Applet!", 140, 50); // Draw text at
(140, 50)
    }

    // Other lifecycle methods (init, start, stop, destroy) could be
added here
    // but are not strictly necessary for this simple drawing example.
}
```

- **Explanation of Code:**
  - We import necessary classes from `java.applet` and `java.awt`.
  - The class `SimpleDrawingApplet` `extends` `Applet`.
  - The special `paint(Graphics g)` method is overridden. The `Graphics` object `g` is provided by the system and is used for all drawing operations.
  - Inside `paint`, we use `g.setColor()` to choose drawing colors and `g.fillRect()` and `g.drawString()` to draw a blue rectangle and red text.

    - The HTML comment block provides instructions and the necessary `<applet>` tag information for the `appletviewer` tool to find and run the applet.
- **Reference to Practical File:**
    - **Program 3:** This program is a more elaborate graphics example in an applet, drawing multiple shapes and colors to create a scene.
    - **Program 4:** This program demonstrates animation within an applet by using a separate thread and repeatedly calling `repaint()` which in turn calls `paint()`.

6. Connecting to a Server / Making URL Connections

- **What is it?**
    - This refers to using the `java.net` package, specifically the `URL` and `URLConnection` (and its subclass `HttpURLConnection`) classes, to interact with resources on the internet, typically web resources. (Notes Image 1 mentions "Connecting to a Server" and "Making URL Connections").
- **Why is it important?**
    - Modern applications often need to fetch data from or send data to web services, APIs, or other servers using standard protocols like HTTP/HTTPS. URL Connections provide a relatively high-level, easy-to-use way to do this without needing to manually handle the low-level details of socket communication for these specific protocols.
- **Key Concepts:**
    - `java.net.URL`: As discussed before, represents the address of a resource (website, file, etc.). (Notes Image 14 & 34 explain it). You create a `URL` object from a String containing the URL address.
    - `java.net.URLConnection`: An abstract class representing a communication link between the application and a URL. You get a `URLConnection` object by calling the `openConnection()` method on a `URL` object.
    - `java.net.HttpURLConnection`: A concrete subclass of `URLConnection` specifically for the HTTP and HTTPS protocols. You typically cast the result of `openConnection()` to this type if you are dealing with HTTP/HTTPS resources, as it provides HTTP-specific methods. (Notes Image 9, 14, 34 mention this).
    - HTTP Methods (GET, POST, PUT, DELETE): `HttpURLConnection` allows you to specify the HTTP method for your request using `setRequestMethod()`. (Notes Image 10 mentions this). GET is for retrieving data, POST is for sending data to be processed.
    - Request Headers: You can set request headers (like `Content-Type`, `Accept`) using `setRequestProperty()`. (Notes Image 33 shows setting `Content-Type`).
    - Sending Data (POST): Requires setting `setDoOutput(true)` and getting an `OutputStream` from the connection to write the data you want to send in the request body. (Notes Images 33 & 35 demonstrate this).
    - Receiving Data (GET or POST Response): Requires getting an `InputStream` from the connection to read the data sent back by the server in the response body. You typically wrap this

stream in a `BufferedReader` to read text data line by line. (Notes Images 11, 12, 13, 33, 35, 39 show getting `InputStream` and using `BufferedReader`).
  - Response Code: You can get the HTTP status code (like 200 OK, 404 Not Found, 500 Internal Server Error) using `getResponseCode()`. (Notes Image 10, 12, 33, 35, 39 show this). This is crucial for knowing if the request was successful.
  - Closing: Call `disconnect()` on the `HttpURLConnection` to close the connection and release resources. Close any streams you opened as well. (Notes Image 12 & 28 mention closing).
- **How it works (Simplified):**
  1. Specify the target resource's address using a `URL` object.
  2. Ask the `URL` object to open a connection (`openConnection()`).
  3. If it's HTTP/HTTPS, work with it as an `HttpURLConnection`.
  4. Configure the request (method, headers, potentially data to send).
  5. Establish the actual connection (implicitly happens when you get streams or response code).
  6. Read the response data from the input stream.
  7. Close the connection.
- **Simple Code Example (Fetching web page content):**

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection; // Specific for HTTP/HTTPS
import java.net.URL; // Represents the URL address

public class SimpleUrlConnectionExample {
    public static void main(String[] args) {
        String urlString = "https://www.example.com"; // The website
address

        try {
            // 1. Create a URL object from the string address
            URL url = new URL(urlString);

            // 2. Open a connection to the URL and cast it to
HttpURLConnection
            HttpURLConnection connection = (HttpURLConnection)
url.openConnection();

            // 3. Set the request method (Optional for GET, but good
practice)
            connection.setRequestMethod("GET"); // Default method is GET
```

```
            // 4. Get the HTTP response code to check if the request was
successful
            int responseCode = connection.getResponseCode();
            System.out.println("HTTP Response Code: " + responseCode); //
Expect 200 for success

            // 5. Check if the response code indicates success (200 OK)
            if (responseCode == HttpURLConnection.HTTP_OK) { //
HttpURLConnection.HTTP_OK is 200
                // 6. Get the InputStream from the connection to read the
response body
                // Wrap it in InputStreamReader and BufferedReader to
read text efficiently
                BufferedReader reader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
                String line;
                StringBuilder content = new StringBuilder();

                // 7. Read the response line by line until the end
                while ((line = reader.readLine()) != null) {
                    content.append(line).append("\n"); // Append the line
and a newline character
                }

                // 8. Close the reader (and implicitly the InputStream it
wraps)
                reader.close();

                // Print the retrieved content
                System.out.println("\n--- Content from " + urlString + "
---");
                System.out.println(content.toString());

            } else {
                System.out.println("GET request failed. Could not
retrieve content.");
            }

            // 9. Close the connection
            connection.disconnect();

        } catch (IOException e) {
            // Handle any errors that occur during the process (e.g.,
```

```
network issues, invalid URL)
            System.err.println("An error occurred during URL
connection:");
            e.printStackTrace();
        }
    }
}
```

- **Explanation of Code:**
  - We define the URL as a string.
  - We create a `URL` object.
  - We call `url.openConnection()` to establish a connection object and cast it to `HttpURLConnection` because we're dealing with HTTP.
  - We explicitly set the request method to "GET" (even though it's the default for many connections).
  - `getResponseCode()` gets the status code (like 200, 404).
  - If the code is 200 (`HttpURLConnection.HTTP_OK`), we get the `InputStream` from the connection.
  - We wrap the `InputStream` first in an `InputStreamReader` (to convert bytes to characters) and then in a `BufferedReader` (to efficiently read lines of text).
  - We read the content line by line using `reader.readLine()` and append it to a `StringBuilder`.
  - Finally, we close the reader and the connection and print the collected content.
  - A `try-catch` block handles potential `IOException`s that might occur during network operations.
- **Reference to Practical File:**
  - **Program 12, 13, 14, 15, 16:** These programs are direct implementations of URL Connections for reading data, sending POST requests, and reading file content via URL, demonstrating the steps outlined above using `HttpURLConnection`, streams, and readers/writers. Program 11 focuses solely on parsing and displaying URL components.

7. Implementing Servers / Socket Programming

- **What is it?**
  - This refers to using the lower-level socket API (`java.net.Socket` and `java.net.ServerSocket`) to create network applications that communicate directly over TCP or UDP protocols. (Notes Image 1 mentions "Implementing Servers" and "Socket Programming").
  - Sockets are endpoints of communication links. Socket programming involves setting up one side (client) to connect to another side (server) at a specific network address (IP) and port.

- **Why is it important?**
  - Provides fine-grained control over network communication.
  - Allows implementation of custom network protocols, not just standard ones like HTTP.
  - Essential for building server applications that listen for and handle connections from multiple clients simultaneously (e.g., chat servers, game servers, custom service backends).
- **Key Concepts:**
  - Client-Server Model: Socket programming heavily relies on this. A **server** runs on a known address (IP + Port) and passively waits for incoming connections. A **client** initiates a connection to a specific server address and port.
  - `java.net.ServerSocket` (Server Side): Represents the server application's endpoint that listens for connection requests from clients.
    - You create a `ServerSocket` bound to a specific port number (`new ServerSocket(port)`). (Notes Images 21, 25).
    - The `accept()` method is crucial. It pauses the server and waits ("blocks") until a client tries to connect. When a connection arrives, `accept()` returns a `Socket` object that represents the dedicated connection to *that specific client*. (Notes Images 21, 23, 26).
  - `java.net.Socket` (Client Side & Server Side): Represents one endpoint of an actual connection.
    - **Client Side:** You create a `Socket` object and specify the server's address and port to connect (`new Socket("host", port)`). (Notes Image 17).
    - **Server Side:** The `ServerSocket`'s `accept()` method *returns* a `Socket` object for each connected client.
  - IP Address: Identifies a specific machine on a network (e.g., `127.0.0.1` for the local machine, "localhost"). (Notes Image 17).
  - Port Number: A number (0-65535) that identifies a specific *process* or *application* running on a machine. Standard services use well-known ports (HTTP 80, HTTPS 443), but you can use higher ports for custom applications (e.g., 5000, 8080). Non-privileged users need ports > 1023. (Notes Image 17, 25).
  - Input/Output Streams: Once a connection is established via a `Socket` object, you get `InputStream` and `OutputStream` from it using `socket.getInputStream()` and `socket.getOutputStream()`. These streams are the channels for sending and receiving raw byte data. (Notes Images 18, 21, 27). You typically wrap these streams with higher-level reader/writer classes (like `BufferedReader`, `PrintWriter`, `DataInputStream`, `DataOutputStream`) to easily work with text or specific data types. (Notes Images 18, 21, 27).
  - Communication: Data is sent by writing to the `OutputStream` and received by reading from the `InputStream`. (Notes Images 18, 21, 27).
  - Closing: It's essential to close the streams and the `Socket` object when the communication is finished to release network resources. On the server side, you close the client `Socket` for each

client, and finally, close the `ServerSocket` when the server application is shutting down. (Notes Images 19, 22, 28).

- **How it works (Simplified):**
  1. **Server:**
     - Creates a `ServerSocket` on a specific port.
     - Enters a loop (or just calls `accept()` once for a simple server), calling `accept()`, waiting for clients.
     - When a client connects, `accept()` returns a `Socket` for that client.
     - Gets input/output streams from the client `Socket`.
     - Communicates with *this specific client* using the streams.
     - Closes the client `Socket` when done with that client.
     - (To handle multiple clients concurrently, each connection typically gets its own thread).
  2. **Client:**
     - Creates a `Socket` object, specifying the server's IP and port. This attempts to connect to the server.
     - Gets input/output streams from its `Socket`.
     - Communicates with the server using the streams.
     - Closes the `Socket` when done.
- **Simple Code Example (Basic Client-Server text exchange):**

  SimpleServer.java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket; // Used on the server side
import java.net.Socket; // Used on both client and server sides

public class SimpleServer {
    private static final int PORT = 5000; // Port number to listen on

    public static void main(String[] args) {
        // Use try-with-resources to ensure sockets are closed
automatically
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("Server started. Listening on port " +
PORT);
            System.out.println("Waiting for a client connection...");
```

```java
            // serverSocket.accept() blocks until a client connects.
            // It returns a Socket object for the connected client.
            Socket clientSocket = serverSocket.accept();
            System.out.println("Client connected: " +
clientSocket.getInetAddress().getHostAddress());

            // Get input and output streams from the client socket
            // Use true for auto-flushing the output stream with println,
printf, or format
            PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

            // Read message from client
            String clientMessage = in.readLine(); // Reads a line of text
            System.out.println("Received from client: " + clientMessage);

            // Send response back to client
            String serverResponse = "Hello back, Client!";
            out.println(serverResponse); // Sends a line of text
            System.out.println("Sent to client: " + serverResponse);

            // Close streams and the client socket when done
            in.close();
            out.close();
            clientSocket.close();
            System.out.println("Client connection closed.");

        } catch (IOException e) {
            System.err.println("Error in server operation: " +
e.getMessage());
            e.printStackTrace();
        }
        System.out.println("Server exiting.");
    }
}
```

SimpleClient.java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
```

```java
import java.net.Socket; // Used on both client and server sides

public class SimpleClient {
    private static final String SERVER_ADDRESS = "localhost"; // Server IP address or hostname
    private static final int PORT = 5000; // Server port number

    public static void main(String[] args) {
        // Use try-with-resources to ensure the socket is closed automatically
        try (Socket socket = new Socket(SERVER_ADDRESS, PORT); // Attempts to connect to the server
             // Get input and output streams from the socket
             // Use true for auto-flushing the output stream
             PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
             BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

            System.out.println("Connected to server at " + SERVER_ADDRESS + ":" + PORT);

            // Send message to server
            String messageToSend = "Hello Server from Client!";
            out.println(messageToSend); // Sends a line of text
            System.out.println("Sent: " + messageToSend);

            // Read response from server
            String response = in.readLine(); // Reads a line of text
            System.out.println("Received: " + response);

        } catch (IOException e) {
            System.err.println("Error in client operation: " + e.getMessage());
            e.printStackTrace();
        }
        System.out.println("Client exiting.");
    }
}
```

- **Explanation of Code:**
  - **Server:**
    - Creates a `ServerSocket` on port 5000.

        - Calls `serverSocket.accept()`, which pauses the program until a client connects to port 5000.
        - When a client connects, `accept()` returns a `Socket` (`clientSocket`) representing that connection.
        - It gets `PrintWriter` and `BufferedReader` from `clientSocket`'s streams to send and receive text lines easily.
        - It reads one line from the client, prints it, then sends one line back as a response, prints what it sent, and closes the connection resources.
    - **Client:**
        - Creates a `Socket` object, attempting to connect to "localhost" (this machine) on port 5000. This line initiates the connection attempt.
        - If successful, it gets `PrintWriter` and `BufferedReader` from its socket's streams.
        - It sends one line of text to the server using `out.println()`.
        - It reads one line of response from the server using `in.readLine()`.
        - It prints the response and closes the connection resources.
    - **To Run:** You must compile both files (`javac SimpleServer.java SimpleClient.java`). Then, *run the server first* (`java SimpleServer`). The server will print "Waiting for a client connection...". *Then, run the client* (`java SimpleClient`). The client will connect, they will exchange messages, and both programs will finish.
- **Reference to Practical File:**
    - **Program 8, 9, 10:** These programs build on this basic client-server interaction, exploring sending/receiving messages in different orders and achieving bidirectional communication.
    - **Notes Images 17-32:** Provide extensive notes, including code snippets, for the client and server implementations, covering socket creation, streams, and the logic for sending/receiving messages (including a loop in the practical server notes to read multiple messages). The notes also introduce a more generic server structure using a separate `handleConnection` method, which is common practice for servers needing to manage multiple clients concurrently (often using threads, although not explicitly shown in the simple generic server code provided in the notes).

8. Java NIO (New Input/Output)

- **What is it?**
    - Introduced in JDK 1.4 to provide more efficient I/O operations, especially for large files or network operations. (Notes Image 15)
    - Includes packages like `java.nio.file`. It offers alternatives to the older `java.io` streams.
- **Why is it important?**
    - Offers features like Channels (fast data transfer), Buffers (efficient data handling), and Selectors (handling multiple connections concurrently, important for high-performance servers).

- The `java.nio.file` package provides a modern, flexible way to interact with the file system.

- **Key Concepts:**
  - `java.io` (Old I/O): Stream-oriented (data flows continuously), blocking I/O (a read/write operation waits until it's complete).
  - `java.nio` (New I/O): Channel-oriented (data read/written to/from buffers), non-blocking I/O is possible (an operation might return immediately, and you check later if it's finished), offers more control over buffers and file access. The notes mention it implements "high speed IO operations". (Notes Image 15).
  - Channels: A connection to an I/O source or destination (like a file or socket). Data is read from/written to Channels into/from Buffers.
  - Buffers: Blocks of memory used to hold data temporarily while it's being transferred via Channels.
  - Selectors: Used in non-blocking I/O to monitor multiple Channels and detect when a Channel is ready for an I/O operation (like reading or writing) without blocking.
  - `java.nio.file.Files`: Contains static utility methods for common file operations (copy, move, delete, read/write all bytes/lines). (Notes Image 15).
  - `java.nio.file.Path`: Represents the location of a file or directory in the file system in a platform-independent way. You get `Path` objects using `java.nio.file.Paths.get()`. (Notes Image 15, 16).

- **Simple Code Example (Using `java.nio.file` to read a file):**

```java
import java.io.IOException;
import java.nio.file.Files; // Utility class for file operations
import java.nio.file.Path; // Represents a file path
import java.nio.file.Paths; // Utility class to get Path objects
import java.util.List; // To hold lines read from the file

public class SimpleNioFileExample {
    public static void main(String[] args) {
        // Define the path to a file (replace with an actual file on your
system)
        // For example, you could create a file named "mydata.txt"
        // Path filePath = Paths.get("mydata.txt");
        // Or a path in a directory:
         Path filePath = Paths.get("data", "mydata.txt"); // Example:
looks for data/mydata.txt

        System.out.println("Attempting to read file: " +
filePath.toAbsolutePath());

        try {
```

```
            // Check if the file exists
            if (Files.exists(filePath)) {
                 // Read all lines from the file into a List of Strings
                 List<String> lines = Files.readAllLines(filePath);

                 System.out.println("\n--- File Content ---");
                 // Print each line
                 for (String line : lines) {
                     System.out.println(line);
                 }
                 System.out.println("--- End of File ---");

            } else {
                System.err.println("Error: File not found at " +
filePath.toAbsolutePath());
                // You might want to create the file if it doesn't exist
                // Files.createFile(filePath);
                // System.out.println("Created new file: " +
filePath.toAbsolutePath());
            }

        } catch (IOException e) {
            // Catch potential errors during file operations
            System.err.println("An error occurred during file reading:");
            e.printStackTrace();
        }
    }
}
```

(Before running this, create a directory named `data` in the same location as your `.java` file, and inside `data`, create a file named `mydata.txt` with some text lines.)

- **Explanation of Code:**
  - We import `Path`, `Paths`, `Files`, and `List` from the `java.nio.file` and `java.util` packages.
  - `Paths.get("data", "mydata.txt")` creates a `Path` object representing the location `data/mydata.txt`. `toAbsolutePath()` converts it to the full path on your system for printing.
  - `Files.exists(filePath)` checks if the file exists at that path.
  - `Files.readAllLines(filePath)` is a simple utility method that directly reads all lines of text from the file into a `List<String>`. This handles opening, reading, and closing the file internally.
  - We iterate through the list and print each line.
  - A `try-catch` block is used to handle potential `IOException`s, such as the file not existing or permission issues.

- **Reference to Practical File:**
    - **Notes Image 15 & 16:** Introduce the `java.nio.file` package, the `Files` utility class, and the `Path` object, explaining their purpose and providing the `Paths.get()` method signature. Program 16 in the practical file uses `HttpURLConnection` to read a file *from a URL*, which is different from reading a local file with `java.nio.file`, but the concept of reading content is related.