GUVI | HCL
Skill Up. Level Up

Java Advanced Topics & Features

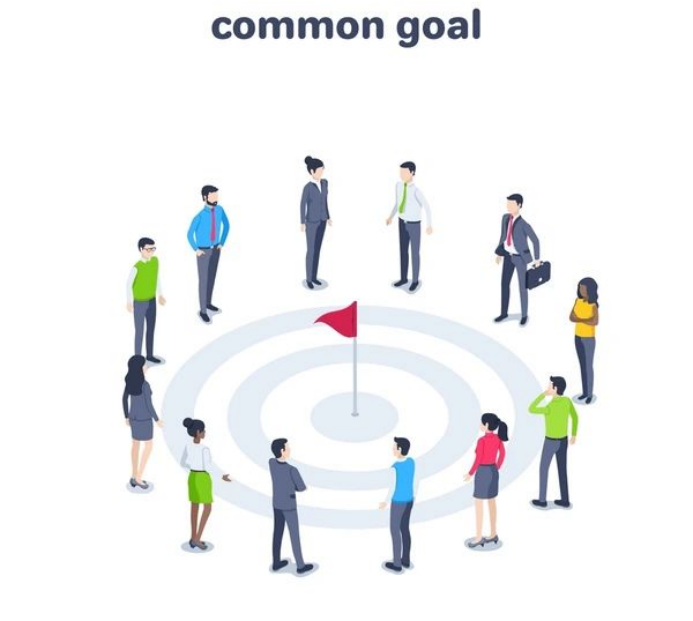
Programming Foundation

Presented by
Nikhil Nair

Website
www.guvi.com

Objectives

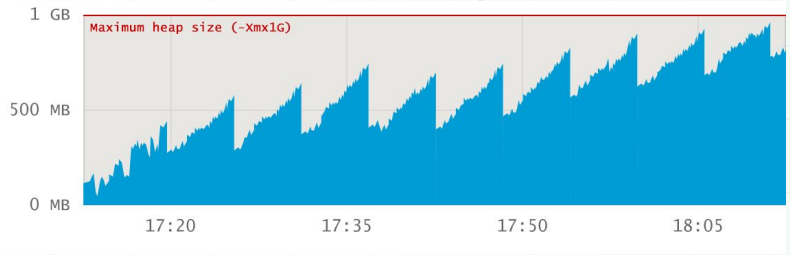
- What you will learn today
    - Memory Management (Stack, Heap, & their uses)
    - Newer, relevant Java Features: var, record, etc
    - Annotations & Enums
    - Internalization



Memory Management

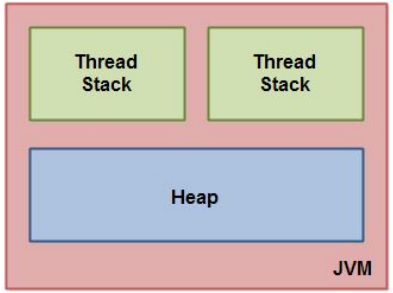
Why programs run out of memory

- Out Of Memory occurs when required allocation exceeds available memory
    - Eg: A request handler appends data to a long lived list on every request
    - Eg: A background job keeps every processed record for later debugging
- Retained references keep objects alive even when the task finished.
    - If an object is still stored in a field or collection it cannot be reclaimed.
- Temporary spikes also exhaust memory when many large objects are created together
    - Eg: Loading ten large files into memory at once; alternative: chunking



Stack and heap in simple terms

- The call stack stores frames for active methods and their locals
    - A frame contains parameters, local primitive values, and reference variables
- Heap is region of memory managed by the JVM, where global objects live
    - Objects are accessed through references stored in stack frames or fields
- Frames are added on call and removed on return using last in first out order
    - When the last reference to an object disappears the object becomes collectible



How Java allocates memory

- On method entry, the JVM creates a stack frame for parameters and locals
    - Primitive locals like int and double live directly in the frame
- The new expression allocates an object on the heap and returns a reference
    - The reference value is stored in a variable within a frame or a field
- Returning from a method discards its frame but not the heap objects it referenced
    - If another live frame or field still holds a reference the object remains alive

```
class Config {
  private int size;
  public Config(int s){ this.size = s; } }
  public static void work() {
    int count = 3;
    Config cfg = new Config(1024);
    byte[] buf = new byte[cfg.size];
    System.out.println(buf.length + ":" + count);
  }
 public static void main(String[] a){ work(); }
}
```

Garbage collection

- Garbage collection is automatic reclamation of heap objects that are no longer reachable
    - Reachable means there is a chain of references from a root to the object
- Roots are active stack frames, static fields, and special native handles
    - If no path from any root exists the object is eligible for reclamation
- A typical cycle marks reachable objects then reclaims the rest in a later phase
    - Short pauses can occur while marking and cleaning happen

```
class GCDemo {
  static byte[] rootHold;
  static void run() {
    byte[] tmp = new byte[1_000_000];
    rootHold = tmp;
    tmp = null;
    rootHold = null;
  }
  public static void main(String[] a){ run(); }
}
```

Why leaks still occur in managed memory

- A Java memory leak is an object that is not needed but remains reachable.
    - Eg: A static list keeps adding tasks and never removes old entries
- Listener and callback references keep publishers and subscribers alive unintentionally.
    - Eg: A view registers a listener then never unregisters it after closing
- Caches can hold on to values forever when no time bound or size bound exists.
    - Eg: A map stores every response by key and the map lives for the process.
- Thread local data can outlive work when threads are reused by libraries.
    - Eg: A value is set on a worker thread and never cleared on completion

```
class LeakDemo {
  static final java.util.List<byte[]> cache = new
java.util.ArrayList<>();
  static void process() {
  cache.add(new byte[100_000]);
  }
  static void simulateWork() {
    for(int i=0;i<10_000;i++) { process(); }
  }
  public static void main(String[] a){
    simulateWork();
    System.out.println(cache.size());
  }
}
```

Activity: Memory simulation

- Scenario
  - You’re writing a small image-processing app.
  - Each image is loaded, processed, and then displayed once
  - However, sometimes the program’s memory keeps growing after every image.
  - You need to reason about what is staying in memory and why
- Tasks
  - Draw what exists on the stack and what exists on the heap while main() runs.
  - Show what happens to the stack and heap after img = null.
  - Explain which object(s) are still reachable and why.
  - Suggest one change that would let the image be garbage-collected after processing.

Activity: Starter Code

```java
import java.util.*;

class Image {
    byte[] data;
    Image(byte[] d){ data = d; }
}
```

```java
public class MemoryCheck {
    static List<Image> processed = new ArrayList<>();

    static Image load() {
        byte[] raw = new byte[512];
        return new Image(raw);
    }

    static void process(Image img) {
        processed.add(img); // keep track of processed images
    }

    public static void main(String[] args) {
        Image img = load();
        process(img);
        img = null;
        System.out.println("done");
    }
}
```

Java’s Newer Features

Why modern Java evolved

- Older code was verbose and repeated simple structural patterns
    - Eg: Plain data classes needed getters, equals, and toString everywhere
- Developers wanted clearer code that kept static types and intent
    - Features reduce boilerplate without changing program behavior or safety
- Changes arrived across versions from Java 10 through Java 21
    - var reduces noisy local types. text blocks remove escape clutter
    - records generate members for data carriers. sealed classes restrict inheritance

Local variable inference with var

- var lets the compiler infer the declared type from the initializer
    - The variable still has a fixed compile time type checked by javac
- Prefer var when the initializer makes the type obvious to humans
    - Avoid var when the expression hides intent or reduces readability

```
class VarDemo {
  public static void main(String[] a){
    var name = "Ada";
    var len = name.length();     // 3
    var ids = new java.util.ArrayList<Integer>();
    ids.add(len);
    for (var id : ids) System.out.println(id);
  }
}
```

Text blocks for readable multiline strings

- A text block is a multiline string delimited by three quotes
    - It preserves newlines and avoids many escape characters for readability
- Incidental indentation can be trimmed so code stays aligned
    - The resulting content matches the visible layout of the block

```
class TextBlockDemo {
  public static void main(String[] a){
    String name = “Nikhil”;
    String sayHi = “Hello “ + name + “. Welcome to the lecture!”
    String json = """
      { "name": "Ada", "id": 1 }
      Test
      """;
    System.out.println(json.contains("Ada"));
  }
}
```

Record classes for pure data

- A record declares a shallowly immutable data carrier with generated members
    - The compiler creates constructor, accessors, equals, hashCode, toString automatically
- Records express that identity equals data, not object identity or behavior
    - Components are final and set at construction for stable state

```
record Point(int x, int y) {}
class RecordDemo {
  public static void main(String[] a){
    var p = new Point(2, 3);
    System.out.println(p.equals(new Point(2, 3)));
    System.out.println(p.x() + p.y());
  }
}
```

Sealed classes to control inheritance

- A sealed type restricts which classes or interfaces can extend or implement it
    - Subtypes must be listed with permits and be final, sealed, or non sealed
- This enables reasoning over a closed set of allowed variants
    - Readers know every legal implementation by inspecting one declaration

```
sealed interface Payment permits Circle, Rect, Square {}
record Circle(double r) implements Shape {}
record Rect(double w, double h) implements Shape {}
record Square(double w, double h) implements Shape {}
class SealedDemo {
  public static void main(String[] a){
    Shape s = new Circle(2);
    System.out.println(s.getClass().getSimpleName());
  }
}
```

Activity concise data design

- Context: you are designing a simple developer profile card for an internal dashboard. The goal is to make the code cleaner and more expressive using modern Java features
- Tasks
    - Replace the Developer class with a record that models name, id, and note.
    - Store the developer’s note as a text block with two lines: greeting and reminder.
    - Use var wherever the initializer makes the type obvious.
    - Redesign Badge as a sealed interface with exactly two permitted types: GoldBadge and SilverBadge.
    - Identify one feature that improves readability but could limit flexibility, and explain why.

Activity: Starter Code

```java
public class Developer {
    private String name;
    private int id;
    private String note;

    public Developer(String name, int id, String note) {
        this.name = name;
        this.id = id;
        this.note = note;
    }

    public String getNote() { return note; }
}
```

```java
public class Badge {
    String type;
    public Badge(String t) { this.type = t; }
}

public class DevDemo {
    public static void main(String[] args) {
        Developer dev = new Developer("Ada", 7, "Welcome to the project!");
        Badge b = new Badge("Gold");
        System.out.println(dev.getNote());
    }
}
```

Annotations and Enums

Why metadata matters

- An annotation is structured metadata you attach to program elements
    - Tools and the compiler can read it to enforce specific rules
- The @Override annotation checks that a method truly overrides a parent
    - If the signature does not match, compilation fails with a clear message
- An enum models a fixed set of named values as a dedicated type
    - Each constant is a predefined value created once and reused by the runtime

Built-in annotation @Override

- @Override verifies that a method signature matches an inherited method
    - Catches typos or wrong parameter lists during compilation
    - Eg: Without it, greet(String) vs greet(String, int) compile silently but behave differently
- Without @Override a misspelled method silently becomes a new unrelated method
    - The program calls the parent method instead of your intended override

```
class Parent {
  void greet(String name) { System.out.println("Hi " + name); }
}
class Child extends Parent {
  @Override
  void greet(String name) { System.out.println("Hello " + name); }
}
```

Creating a Custom Annotation

- Declaration: Use @interface to define a new annotation type
    - Fields (eg: String value()) are optional & permit annotations carry data like configuration values, labels, etc
- Retention policy: Use @Retention to specify how long the annotation should be retained
    - SOURCE: Discarded during compilation
    - CLASS: Present in .class files, not at runtime
    - RUNTIME: Available at runtime via reflection
- Specify target: Use @Target to define where the annotation can be applied. Eg: METHOD, FIELD, TYPE, PARAMETER, etc
- Apply the annotation: Use it in your code to mark elements (e.g., @MyAnnotation("info")).
- Process it: Use reflection (isAnnotationPresent, getAnnotation) to inspect and act on annotations at runtime

```
// Annotation declaration
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyAnnotation {
    String value() default "info";
}
// Annotation usage
public class Example {
    @MyAnnotation("test")
    public void doSomething() {}
}
// Using Reflection to inspect the annotation
Method m = Example.class.getMethod("doSomething");
if (m.isAnnotationPresent(MyAnnotation.class)) {
    String val = m.getAnnotation(MyAnnotation.class).value();
    System.out.println("Annotation value: " + val);
}
```

Enum types in action

- Special data type that represents a fixed set of constants
    - Eg: NEW, IN_PROGRESS, DONE
- Improves type safety compared to using plain int or String constants
    - The compiler prevents invalid values and typos
- Defined using the enum keyword

```
enum Status {
  NEW { String label() { return "New"; } },
  IN_PROGRESS { String label() { return "In progress"; } },
  DONE { String label() { return "Done"; } };
  abstract String label();
}
class Board {
  static void print(Status s){ System.out.println("Status " + s.label()); }
  public static void main(String[] a){ print(Status.NEW); }
}
```

Built-in annotation @Override

- An enum defines a closed vocabulary for a concept in your program
    - Values are type checked and cannot be mistyped like free form strings
    - Use cases: days of the week or traffic lights are given the fixed, known set of values
- Enum constants can carry data and behavior with fields and methods
    - You can add a method to format a label or compute next state

```
enum Status {
  NEW { String label() { return "New"; } },
  IN_PROGRESS { String label() { return "In progress"; } },
  DONE { String label() { return "Done"; } };
  abstract String label();
}

class Board {
  static void print(Status s){
    System.out.println("Status " + s.label());
  }
  public static void main(String[] a){
    print(Status.NEW);
  }
}
```

Activity: Annotation and enum combo

- Context: simple notification system manages messages of different types but currently uses plain strings and has an override error
- Tasks
    - Add @Override in the subclass and fix the method signature so overriding works correctly
    - Replace the fragile string type with an enum Channel having values EMAIL, SMS, and PUSH
    - Add a method label() in Channel to return a friendly name such as "Email message"
    - Use the enum in main() to print the readable channel label after sending the message

```
class Notification {
    String type; // "EMAIL", "SMS", "PUSH"
    void send(String msg){ System.out.println("Sending: "
+ msg); }
}

class EmailNotification extends Notification {
    void Send(String msg){ System.out.println("Email: " +
msg); } // wrong case
}

public class NotifyDemo {
    public static void main(String[] a){
        Notification n = new EmailNotification();
        n.send("Hello user!");
    }
}
```

Internationalization

Intro to Internalization & Locale

- What's internalization?
    - The process of designing an application so it can easily support multiple languages, regions, and cultural formats
    - Can include changes to text, dates, numbers, and currencies & more without changing the source code
- In Java, A Locale identifies a user's language and region for formatting
    - Eg: English (United States), Hindi (India)
- Create a Locale from tags or explicit parts
    - Locale.forLanguageTag("en-US"), new Locale("hi", "IN")
- Locale is a selector for language-aware APIs at runtime
    - Pass it when looking up a ResourceBundle

```
import java.util.Locale;

public class LocaleDemo {
  public static void main(String[] args) {
    Locale enUS = Locale.forLanguageTag("en-US");
    Locale hiIN = new Locale("hi", "IN");
    System.out.println(enUS.getLanguage() + " " +
enUS.getCountry());
    System.out.println(hiIN.getLanguage() + " " +
hiIN.getCountry());
  }
}
```

ResourceBundle for externalized messages

- An ResourceBundle maps keys to texts chosen by Locale
    - Files share a base name and add language or region suffixes
- Lookup chooses the best match for the given Locale
    - Falls back to default bundle if a specific one is missing
- Keep key names consistent across all bundle files
    - Missing keys cause errors during lookup

```
import java.util.*;

public class BundleDemo {
  public static void main(String[] args) {
    Locale hiIN = new Locale("hi", "IN");
    ResourceBundle rb =
ResourceBundle.getBundle("Messages", hiIN);
    System.out.println(rb.getString("greeting"));
  }
}

// Resource file
Messages.properties
Messages_en_US.properties
Messages_hi_IN.properties
```

Activity: Locale switcher with enum mapping

- Context. A console app prints a welcome and a short help note, both hardcoded in English
- Tasks
    - Create Messages.properties and one more language file with the same keys: greeting, help.
    - Read a Locale from a code constant, not the system default.
    - Load the ResourceBundle and print greeting and help from the bundle.
    - Add a bundle key channel.EMAIL (and SMS, PUSH). Print a localized channel label.
    - Ensure a safe fallback when a channel label key is missing in a non-default bundle.

```
enum Channel { EMAIL, SMS, PUSH }

public class HelloApp {
  public static void main(String[] args) {
    Channel c = Channel.EMAIL;
    String greeting = "Welcome to our application";
    String help = "Type H for help, Q to quit";
    System.out.println("Channel: " + c);
    System.out.println(greeting);
    System.out.println(help);
  }
}
```

Localized enum labels in practice

- Enums and bundles work together to keep UI text out of code
    - Use enum names as bundle key suffixes for simple lookup
- Compose output strings using only bundle texts and enum-based keys
    - Avoid concatenating translatable words in code

```
import java.util.*;

enum Status { NEW, IN_PROGRESS, DONE }

public class StatusLabels {
  public static void main(String[] a){
    Locale chosen = Locale.forLanguageTag("en-US");
    ResourceBundle rb =
ResourceBundle.getBundle("StatusTexts", chosen);
    Status s = Status.IN_PROGRESS;
    String key = "status." + s.name();
    System.out.println(rb.getString(key));
  }
}
// Resource file
StatusTexts.properties
StatusTexts_en_US.properties
StatusTexts_hi_IN.properties
```

That’s for today!
Any questions?