GUVI | HCL
Skill Up. Level Up

Control Flow, Loops & Functions

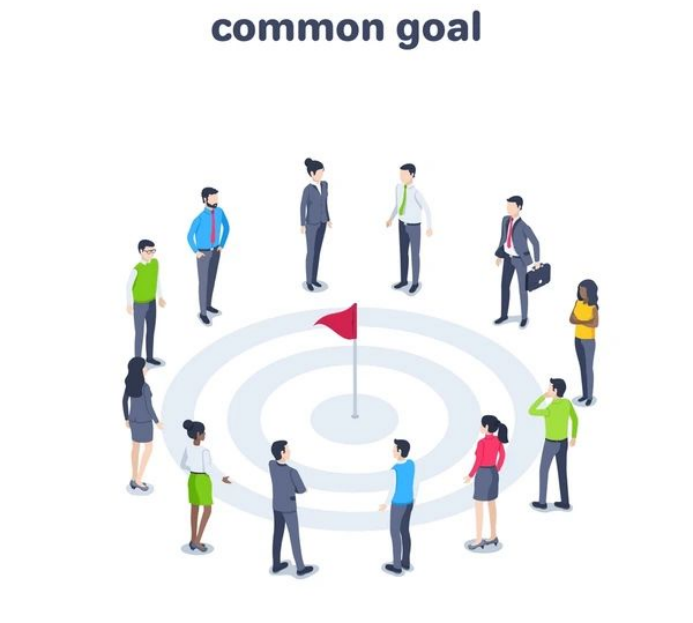
Programming Foundation

Presented by
Nikhil Nair

Website
www.guvi.com

Objectives

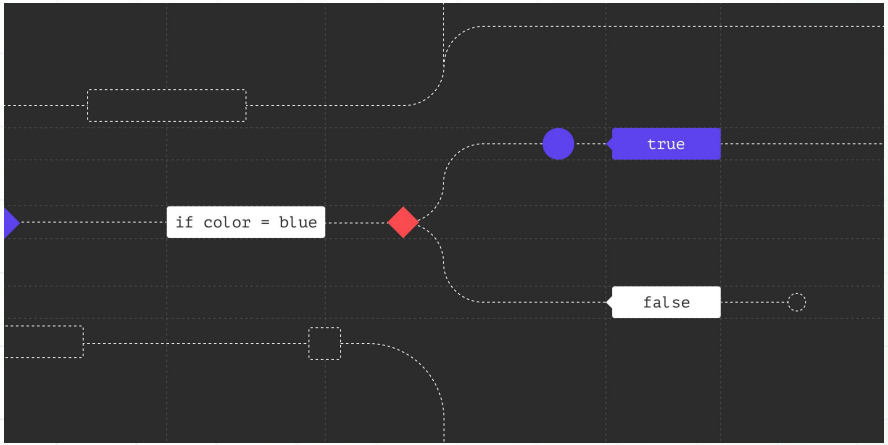
- An introduction to conditionals
- Discovering looping constructs
- Understanding functions



Conditionals

What is a conditional?

- A **conditional** is a rule that evaluates to true or false
    - Allows the program to choose exactly one path to execute next
- In Java, only a boolean expression can control the decision; numbers or text are not treated as “truthy” or “falsy.”
- We write the rule as a test (eg, “hour >= 7”) and the language runs one block if the test is true, otherwise another



Operators: Comparison (relational)

- **Operators** are symbols that perform operations on values
- **Comparison operators** answer questions and *return* a boolean
- The operators include <, <=, >, >=, == (“is equal to”), and != (“is not equal to”); we read them as sentences to avoid mistakes
- Examples
    - “score is at least 60” -> score >= 60
    - “temperature equals 100” -> temperature == 100
    - letter is not ‘A’ -> letter != ‘A’

Operators: Logical

- **Logical operators** work on booleans:
    - && (and) means both conditions must be true
    - || (or) means at least one condition is true
    - ! (not) flips a boolean
- Java **short-circuits**:
    - in A && B, if A is false then B is not evaluated
    - in A || B, if A is true then B is not evaluated
- Examples
    - “pass if score >= 60 and attendance ≥ 75%” => (score >= 60) && (attendance >= 75)
    - “apply discount if user is a member or cart total >= 500” -> isMember || (cartTotal >= 500)

Designing Boundaries

- A boundary rule must say whether the limit counts or doesn’t count; this choice changes edge outcomes
    - “Pass at 60 or more” => score >= 60
    - “Pass at more than 60” => score > 60
- Always sanity-check values just below, at, and just above the limit so the rule behaves as intended
    - For score >= 60,
        - Think “59 => fail; 60 => pass, 61 => pass”
- Write the English rule first, then the boolean; keep them aligned to avoid silent mistakes

If-Else Chains: Exactly One Branch Runs

- Structure:
    - if (condition) { … }: handle the first matching case
    - else if (nextCondition) { … }: handle the next specific cases in order
    - else { … }: handle everything not matched above
- Make ranges non-overlapping and place more specific checks before broader ones; otherwise later branches never run
- Give each branch one clear responsibility (for example, “assign grade” or “show ‘invalid input’ message”), keeping the ladder easy to read

```
if (score >= 90) { grade = "A"; }
else if (score >= 80) { grade = "B"; }
else if (score >= 60) { grade = "C"; }
else { grade = "D"; }
```

Classic switch Statement: case, break, and default

- **switch (value)**
  - evaluates the condition once
  - compares to labeled cases
  - and chooses the matching block
  - default handles “anything else”
- **Break;**
  - ends the chosen block;
  - without it, control falls through into the next case
  - use explicit *break;* when you intend only one block to run
- Use for **discrete choices** (eg: day number, menu option) where each label has a short, clear action or assignment

```
switch (day) {
  case 1: name = "Mon"; break;
  case 2: name = "Tue"; break;
  case 3: name = "Wed"; break;
  default: name = "Unknown";
}
```

From Classic switch to Switch Expression

- With Java 17, a more concise version of Switch statements was released
- Improvements
    - Returns a value directly => results in fewer temporary variables to store
    - Arrow syntax prevents fall-through bugs
    - Encourages exhaustive handling (we'll cover this in lecture 6).
    - default handles "anything else"

```
String grade = switch (score / 10) {
  case 10, 9 -> "A";
  case 8     -> "B";
  case 7     -> "C";
  case 6     -> "D";
  default    -> "F";
};
```

Activity A: Grade Calculator

- **Goal**: Convert a numeric score (0 - 100) to "A" | "B" | "C" | "D" | "F" using clear boundaries, then refactor the mapping to a switch expression
- **Boundaries**: A: 90-100, B: 80-89, C: 70-79, D: 60-69, F: 0-59
    - Treat anything outside 0–100 as invalid
- **Consider**:
    - Edge cases: each of these - 60, 70, 80, 90 - must land in the correct bucket
    - invalid inputs show "Invalid"

Loops

An introduction to Loops

- A loop repeats work zero or more times and is defined by three parts you can name in English:
    - the start (initial values),
    - the condition (when to continue),
    - and the update (how to move toward stopping)
- Example rule: “Start at 1, continue while <= 5, add 1 each time” => runs 5 times
- Changing any one part changes total iterations; a missing or wrong update is the usual cause of an infinite loop
    - Always ask: “What makes this loop eventually stop?”
- Tie the rule to code so intent stays clear:

```
int i = 1;                  // start
while (i <= 5) {            // condition
  // work
  i = i + 1;                // update
}
```

Choosing the right Loop

- We'll cover three looping constructs in Java
- Use **for-loop** when the number of iterations is known or counted; its header shows start, condition, and update together, which makes intent easy to read
    - Example: "Process items 0 to 10" => for (int i = 0; i < 10; i++) { … }
- Use **while** when you repeat until a rule is satisfied and the count is not known up front; it expresses "keep going while this remains true.
    - Example: "Ask until the input is valid" => while (!isValid) { … }
- Use **do-while** when the body must run at least once before you can sensibly check the condition.
    - Example: "Show menu once, then ask to continue" => do { … } while (choice != 'q');

for-loop: Patterns and Off-by-One Decisions

- **Counting up** (0-based): run exactly n times; i takes 0…n-1. Choose < n for the end test.
    - for (int i = 0; i < n; i++) { /* i = 0,1,2,...,n-1 (n times) */ }
- **Counting down** (reverse): be explicit about start and stop to match intent.
    - for (int i = n - 1; i >= 0; i--) { /* i = n-1,...,0 */ }
- Stride & inclusive ends: make the step visible and decide if the end counts (<=) or doesn't (<)
    - for (int i = 0; i <= 100; i += 10) { */* 0,10,20,...,100 (includes 100) */* }
    - for (int i = 1; i < 6; i++) { */* 1,2,3,4,5 (5 times, end excluded) */* }
    - // Sanity check a loop by listing the first few values and the last value you expect.

While-loop

- A **while loop** expresses “**repeat until this rule is satisfied**,” which is ideal for input validation and for searching when the number of iterations is not known in advance
- **Validation pattern**: keep checking the condition until the value passes the rule
    - Use **continue** to skip the rest of an iteration when the current value is unusable

```
boolean valid = false;
while (!valid) {
  // read input here
  if (!isNumeric(input)) { continue; }      // skip this round, try again
  int score = parseInt(input);
  valid = (0 <= score) && (score <= 100);
}
```

do-while: Run Once, Then Decide

- A **do-while** loop runs the **body first**, then checks the condition
- Use it when the action must happen at least once. Eg: show a menu, play one round, then ask to continue
- **Contrast with while**: if the condition is false at the start, a while loop runs zero times, but a do-while still runs once to gather the information needed for the next decision.

```
char choice;
do {
  // show menu and handle one selection
  // choice = readChoice();
} while (choice != 'q');   // keep going until the user chooses to quit
```

Activity B: Number Guessing

- **Goal**
    - The program stores a secret in 1-100 (hardcoded for now)
    - It repeatedly guesses the secret
    - After each guess, it prints “Higher than …” or “Lower than …” by comparing the guess with the secret
    - Once the secretis hit, it stops immediately
- **Loop design:** Use while (true) with an early break on success.
- **Consider**
    - (Optional) It terminates after 20 incorrect guesses
    - Final message prints “Correct in X tries!” where X is the actual attempt count

Activity C: Pattern Printing

- **Goal**:
    - Print a left-aligned triangle of * using a variable rows set at the top (no user input yet)
    - The outer loop controls rows; the inner loop prints row number of stars
- **Design rules**: Decide if the last row counts (we'll make it inclusive). Print a newline after each row; do not hardcode star counts
- **Consider**:
    - Declaring a variable, totalRows, and assigning it a value
    - for-loop where i starts at 1 and goes up to the totalRows, incrementing by 1
    - For each iteration of i, a nested loop to perform the printing

Functions

What is a Function?

- A **function** is a named unit of work: it accepts an inputs, performs some work, and may return an output
- Functions exist to make programs **readable** (names tell the story), **reusable** (no copy-paste), and **testable** (small, single-responsibility pieces)
- Usually, a good function does one thing you can describe in one sentence

```
// What does this method do?
public static int max(int a, int b) {
  return (a >= b) ? a : b;
}
// Called from main: int m = max(3, 5); // m = 5
```

Anatomy of a Function

- The key parts of a function include, **access**, **modifier**, **return type**, **name**, and **parameters**
- Access control who can call a function:
    - May be **public** (anywhere), **private** (only inside the same class), **protected** (package + subclasses) or package-private (if no keyword is provided). Choose the smallest access that works.
- Modifiers: keyword that changes how a method can be called, or how it behaves when the program runs
    - **Static**: a static function belongs to the class and can be called without creating an object (we'll cover this in lecture 6)
- The return type tells callers what value they get back (or void if nothing is returned).
- *Note*: returning a value is different from printing to console; printing returns no value to the caller

```
public class MathUtil {
  public static int max(int a, int b) { return (a >= b) ? a : b; }          // public + static
  private static boolean isNonNegative(int n) { return n >= 0; }      // private helper
}
```

Parameters & Arguments

- **Parameters** are the names and types a function declares as its inputs.
- The actual values passed when a function is called are **arguments.**
- **A function Signature** = function name + parameter types & order
- Question: what if you've multiple methods with the same name?
    - If two methods with the same name exist, the parameter types and order matter help determine which method is invoked

```
static String greet(String first, String last) {
  return "Hello, " + first + " " + last;
}
System.out.println(greet("Ada", "Lovelace"));
// Wrong order gives wrong output, not a compiler error!
```

Overloading

- **Overloading** lets us use the same method name for closely related work, but it declares different parameter lists
- Generally, it's recommended to keep overloads few and unambiguous; provide distinct function names if behaviors start to diverge
- Note: overloading by return type alone isn't permitted in Java, since it's not part of the signature

```
static void f(long x)      { System.out.println("long"); }     // widening
static void f(int x)   { System.out.println("int"); }  // boxing
f(5); // prints "int"
```

That’s for today!
Any questions?