GUVI | HCL
Skill Up. Level Up

Exception Handling

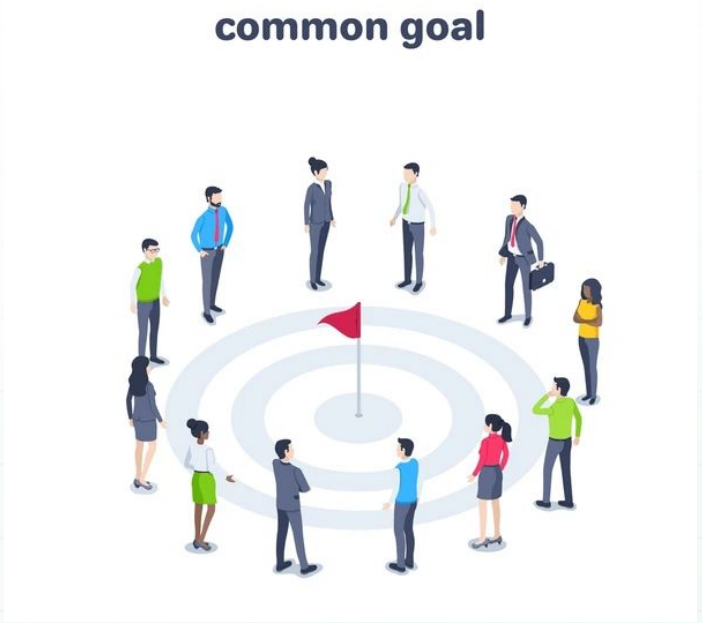
Programming Foundation

Presented by
Nikhil Nair

Website
www.guvi.com

Objectives

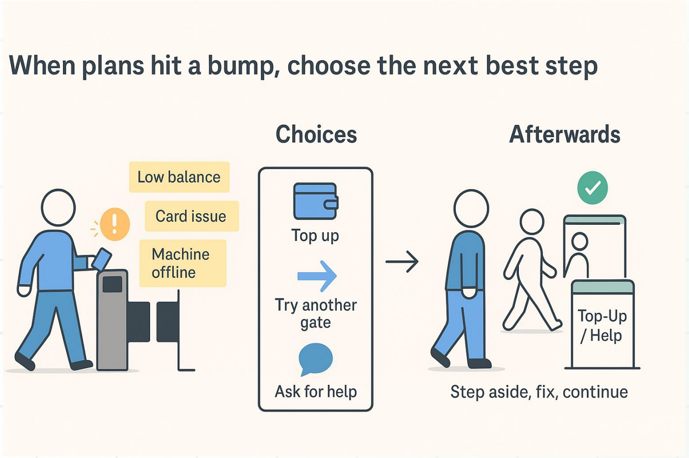
- What you will learn today
  - Purpose of exceptions in programming
  - Checked vs Unchecked exceptions
  - Handling exceptions in Java
  - Custom Exceptions
- How this builds on previous lectures
  - Any methods or static methods may throw exceptions
  - Exceptions can be produced when performing arithmetic operations, user input is invalid, etc



Exceptions

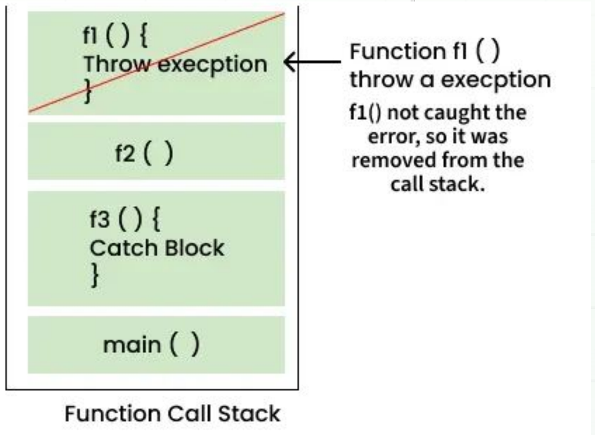
Why exceptions exist

- Inputs, computations, or environment can fail at runtime
- Exceptions provide one structured path for handling failures
- Centralized handling keeps normal logic readable and testable
- Magic return values hide bugs and confuse control flow



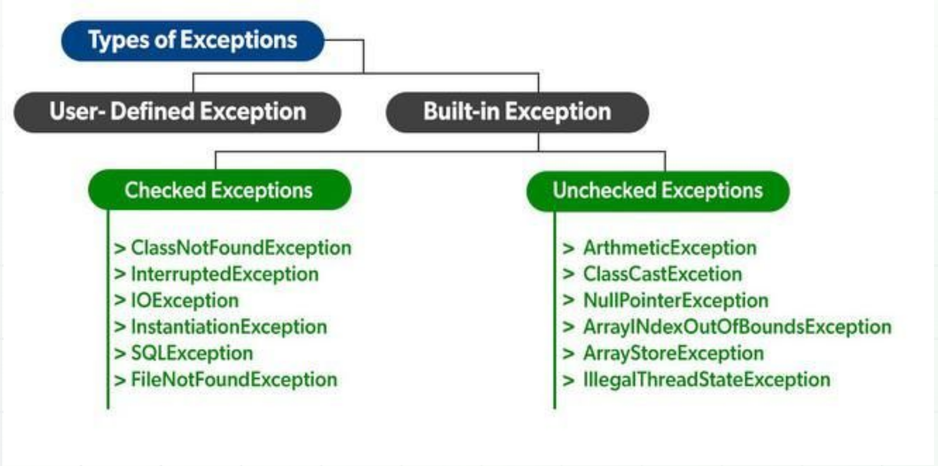
Exception as a concept

- An exception is an object describing a specific problem
- Stack unwinding pops frames until a matching handler appears
- A handler is a catch block that decides action
- Errors mark serious conditions usually outside application control



Checked versus unchecked

- Checked exceptions require handling or declaration at compile time
- Unchecked exceptions occur without compiler enforcement and caller obligations
- Use checked when callers can realistically recover from failure
- Use unchecked for programmer mistakes and invalid API usage



When to use exceptions

- Use exceptions for rare, disruptive events beyond normal expectations
  - Examples: database down, corrupted input, invariant breach, configuration missing
- Prefer return-based validation for expected mistakes and routine branches
  - Examples: empty field, optional filter, retry prompt, missing optional data
- Throw when method contract fails after reasonable local recovery attempts
- Avoid exceptions for loop exits, boolean checks, predictable branches

```java
public static boolean validAge(int age) { return age >= 0; }
public static void requireNonNegative(int n) {
  if (n < 0) throw new IllegalArgumentException("negative not allowed");
}
requireNonNegative(-1);
```

Activity: classify failures

- Classify each scenario as checked or unchecked with a reason checked or unchecked with justification
- Integer.parseInt("abc")
    - unchecked
- a[10] when size three
    - unchecked
- new FileInputStream("missing-file.txt")
    - checked
- Exceptions that must be declared in the method signature
    - checked
- obj.toString() on null
    - unchecked

try and catch anatomy

- A try block encloses code that might throw an exception
- A catch block names the exception type and a variable
- Order multiple catches from specific types to broader base classes
- Continue after catch or finally, unless rethrow or fatal error

```
try {
  int x = Integer.parseInt(s);
  System.out.println(x);
} catch (NumberFormatException e) {
  System.out.println("invalid number: " + e.getMessage());
}
```

Ordering and specificity

- Catch specific exception types before their broader superclasses
- Place broad catch blocks last to avoid unreachable handlers
- Handle distinct failure cases separately to keep recovery actions clear
- Avoid swallowing exceptions by logging context and rethrowing when needed

```
try { Integer.parseInt(s); }
catch (Exception e) { System.out.println("broad"); }
catch (NumberFormatException e) { System.out.println("specific"); } // unreachable
```

Finally semantics

- A finally block runs after matching catch or normal exit
- Finally executes even with thrown exceptions or early returns
- Use finally for essential cleanup like closing scanners (when reading files) or resources (closing a database connection)
- Avoid changing return values or throwing new exceptions from finally

```
static int f() {
  try { return 1; }
  finally { System.out.println("cleanup"); }
}
```

Multi-catch and rethrow

- Multi-catch handles several exception types with identical recovery code
- Use it when messages and actions are truly the same
- Rethrow when this method cannot decide safe recovery behavior
- Prefer translating to domain exceptions for clearer contracts and callers

```
try { Integer.parseInt(s.trim()); }
catch (NumberFormatException | NullPointerException e) {
  throw new IllegalArgumentException("invalid number input", e);
}
```

Handler side effects

- Choose handler action: recover now, translate, or rethrow
- Log only context needed to reproduce and debug failures
- Avoid empty catches that swallow errors and hide bugs
- Prefer specific messages and actions per distinct failure type

```java
try { readConfig(); }
catch (IOException e) {
  System.err.println("config issue: " + e.getMessage());
  throw new IllegalStateException("required config missing", e);
}
```

Activity: try-catch-finally order

- Predict execution order across try, catch, finally, return
- Focus on first matching catch and guaranteed finally run

```java
static int g(String s) {
  try {
    System.out.println("try");
    Integer.parseInt(s);
    return 10;
  } catch (NumberFormatException e) {
    System.out.println("catch");
    return 20;
  } finally {
    System.out.println("finally");
  }
}
// call: g("abc");
```

Activity: try-catch-finally order

```java
static int h() {
  try {
    System.out.println("try");
    return 1;
  } finally {
    System.out.println("finally");
  }
}
// call: h();
```

```java
static void k() {
  try {
    System.out.println("try");
    throw new RuntimeException("x");
  } catch (RuntimeException e) {
    System.out.println("catch");
    throw e;
  } finally {
    System.out.println("finally");
  }
}
// call: k();
```

throw keyword purpose

- Use throw to signal a detected problem immediately
- Create specific exceptions with messages explaining context clearly
- Throw nearest to detection after minimal local checks
- Document thrown types so callers know their obligations

```
if (age < 0) {
   throw new IllegalArgumentException("age must be nonnegative");
}
```

throws and contracts

- throws declares checked exceptions a method may pass to callers
- Callers must handle those exceptions or declare them again
- Do not list unchecked exceptions in a throws clause
- Declare only types the caller truly needs to know

```
public static void enrollStudent(String s) throws Exception {
  int age = Integer.parseInt(s);       // unchecked if bad
  requireValidAge(age);               // checked, must handle or declare
}

public static void requireValidAge(int age) throws Exception {
  if (age < 0) throw new Exception("age cannot be negative");
}
```

Propagation and translation decisions

- Propagate in the middle layers; handle at boundaries; throw at source
- Translate = wrap low-level exception into clearer one, keep original cause
- Do not translate when caller needs the original exception type
- Preserve stack trace by rethrowing or passing the cause to the new exception

```
void runGate(String input) {
  try {
    int fare = parseFare(input);          // may translate (bad input)
    computeFare(fare);                    // may propagate (checked failure)
    System.out.println("entry allowed");
  } catch (IllegalArgumentException e) {  // translated unchecked
    System.out.println("invalid input: " + e.getMessage());
  } catch (Exception e) {                 // propagated checked
    System.out.println("system error: " + e.getMessage());
  }
}

int parseFare(String s) {
  try { return Integer.parseInt(s.trim()); }
  catch (NumberFormatException e) {
    throw new IllegalArgumentException("fare must be a whole number", e); // translate
  }
}

void computeFare(int fare) throws Exception { // propagate
  if (fare <= 0) throw new Exception("fare rule misconfigured");
}
```

Activity: propagate or translate

- A) Input text is "29x" at gate keypad during entry. Assume input must be a number.
    - Translate to IllegalArgumentException with message “fare must be a whole number”
- B) Card balance is lower than fare when tapping gate.
    - Propagate InsufficientBalanceException to boundary for user-facing message
- C) Application startup fails to load required rules for fare from config.
    - Handle here by failing fast with clear error and halt startup

Custom exception roles

- Name domain rules with clear, specific custom exception types
- Separate detection sites from handling sites using those types
- Prefer one type per rule family, not per message variant
- Write actionable messages including failed value and expected constraint

```
public void debit(int bal, int amt) throws InsufficientBalanceException {
  if (amt > bal) throw new InsufficientBalanceException("amount " + amt + " exceeds " + bal);
}
```

Naming and hierarchy

- Choose checked when callers can reasonably recover in context
- Extend Exception for checked, RuntimeException for unchecked
- Use informative names like InsufficientBalance or InvalidPin
- Provide constructors for message and cause preservation

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

Activity: design two exceptions

- Define two domain exceptions with name, type, and usage site
- Pick checked or unchecked based on caller recovery ability
- Write one actionable message with failed value or constraint
- A) Wallet debit fails when amount exceeds available balance.
    - InsufficientBalanceException; checked; thrown in debit step; message “amount 750 exceeds balance 500”
- B) PIN input contains letters instead of digits.
    - InvalidPinException; unchecked; thrown at PIN parse; message “PIN must be four digits 0-9”

Wrap up and next steps

- Recognize when to use, catch, throw, and declare exceptions
- Apply ordering, finally, and multi catch with clear intent
- Decide propagate versus translate based on caller responsibilities
- Design lean custom types with names, hierarchy, and messages

That’s for today!

Any questions?