



## *Exceptions and Exception Handling*

Unit 12



## *Unit Objectives*

---

- To learn how
  - Exceptions are used to signal errors
  - To use `try` and `catch` to handle exceptions
  - To throw exceptions
- To understand assertions and how to use them

## Exceptions

---

- Exception
  - An event or condition that disrupts the normal flow of execution in a program
  - The condition causes the system to **throw** an exception
  - The flow of control is interrupted and a handler will **catch** the exception
- Exception handling is object-oriented and
  - Encapsulates unexpected conditions in an object
  - Provides an elegant way to make programs robust
  - Isolates abnormal from regular flow of control

```
float sales = getSales();
int staffsize = getStaff().size;
float avg_sales = sales/staffsize;
System.out.println(avg_sales);
```

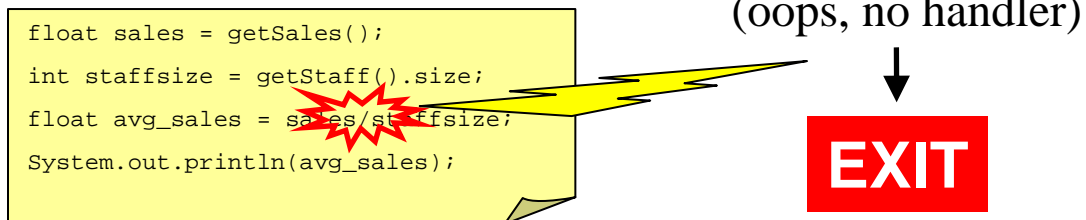


handler

## Exception Sources

---

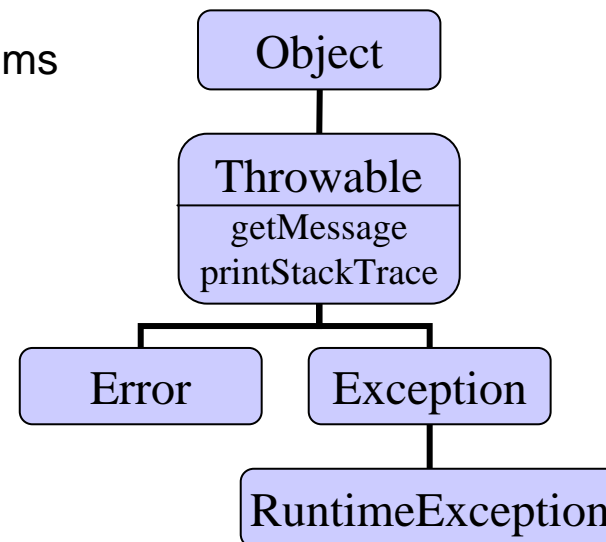
- JVM can detect unrecoverable conditions
  - Examples:
    - Class cannot be loaded
    - null object reference used
- Both core classes and code that you write can throw exceptions
  - Examples:
    - IO error
    - Divide by zero
    - Data validation
    - Business logic exception
- Exceptions terminate execution unless they are handled by the program



## The Exception Hierarchy

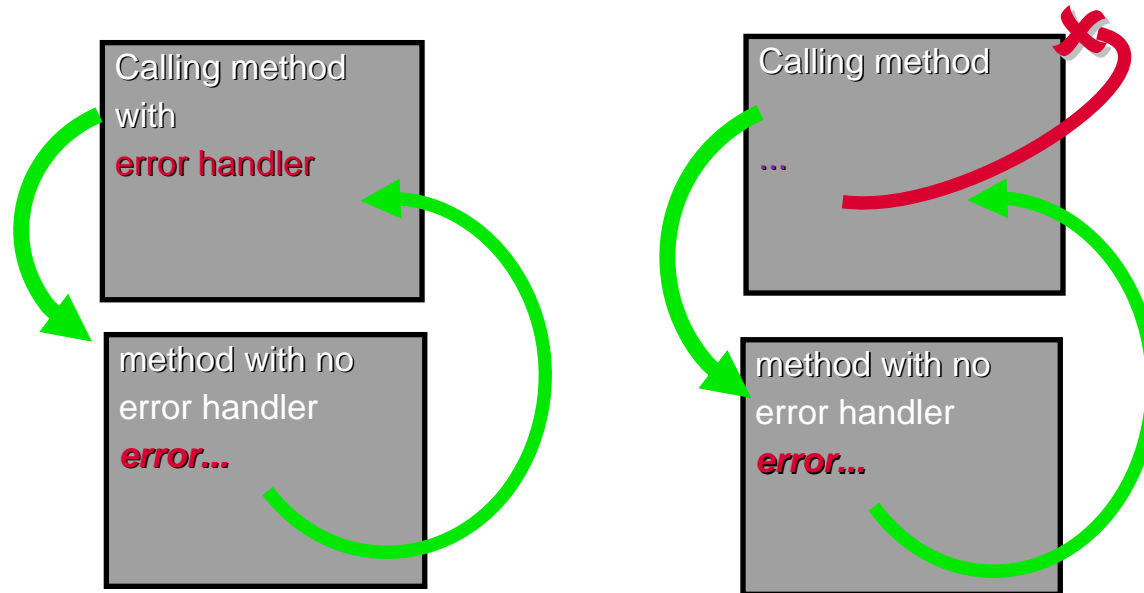
---

- **Throwable** is the base class
  - Provides a common interface and implementation for most exceptions
- **Error** indicates serious problems that should not be caught
  - **VirtualMachineError**
  - **CoderMalfunctionError**
- **Exception** heads the class of conditions that should be caught or specified as thrown
- A **RuntimeException** can be thrown during the normal operation of the JVM
  - **ArithmeticException**
  - **BufferOverflowException**



## Handling Exceptions

- Checked exceptions must either be handled in the method where they are generated or delegated to the calling method:



## Keywords

---

- `throws`
  - A clause in a method declaration that lists exceptions that may be delegated up the call stack
    - e.g.: `public int doIt() throws SomeException, ...`
- `try`
  - Precedes a block of code with attached error handlers – errors in the `try` block are handled by the error handlers
- `catch`
  - A block of code to handle a specific exception
- `finally`
  - An optional block which follows `catch` clauses
  - Always executed regardless of whether an exception occurs
- `throw`
  - Launches the exception mechanism explicitly
    - e.g.: `throw (SomeException)`

## *try/catch Blocks*

---

- To program exception handling you must use `try/catch` blocks
- Code that might produce a given error is enclosed in a `try` block
- The `catch` clause must immediately follow the `try` block

```
try{
    //code that reads input from a file
}
catch (IOException ioe){
    // some code that deals with i/o problems
}
```

## *catch Clauses*

---

- The clause always has one argument that declares the type of exception to be caught
- The argument must be an object reference for the class **Throwable** or one of its subclasses
- Several `catch` clauses may follow one `try` block

```
catch (MyException me) {  
    ...  
}
```

## *try/catch Cont'd*

---

- Java provides an elegant solution to error management with `try/catch` blocks

- Allows you to write the main flow of your code and deal with the exceptional cases later

- Does not spare you the effort of doing the work of detecting, reporting, and handling errors

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    }
    catch (fileOpenFailed) {
        doSomething; }
    catch (sizeDeterminationFailed) {
        doSomething; }
    catch (memoryAllocationFailed) {
        doSomething; }
    catch (readFailed) {
        doSomething; }
    catch (fileCloseFailed) {
        doSomething; }
}
```

## Example

---

```
class MultiCatch {  
  
    public static void main( String args[]) {  
  
        try {  
            // format a number  
            // read a file  
            // something else...  
        }  
        catch( IOException e) {  
            System. out. println(" I/O error " + e.getMessage());  
        }  
        catch( NumberFormatException e) {  
            System. out. println(" Bad data " + e.getMessage());  
        }  
        catch( Throwable e) { // catch all  
            System. out. println(" error: " + e.getMessage());  
        }  
  
    }  
  
}
```

## *The finally Clause*

---

- Optional clause that allows clean-up and other operations to occur whether an exception takes place or not
  - May have `try/finally` with no `catch` clauses
- Executed after any of the following:
  - A `try` block completes normally
  - A `catch` clause executes
    - Even if `catch` clause includes `return`!
  - An unhandled exception is thrown, but before execution returns to calling method

```
try {  
    // file processing  
}  
catch (IOException e) {  
    // handle exception  
}  
finally {  
    // close files  
}
```

## Example

---

```
public class MyTest {  
  
    public static void main(String[] args) {  
        System.out.println(exceptionTest());  
    }  
    public static int exceptionTest() {  
  
        try {  
            int x = 0;  
            int y = 1;  
            System.out.println("Result=" + (y/x));  
        } catch (Exception e) {  
            e.printStackTrace();  
            return 1;  
        } finally {  
            System.out.println("Finally!");  
        }  
        return 2;  
    }  
}
```

## Nested Exception Handling

- May be necessary to handle exceptions when already handling them (i.e., inside a `catch` or `finally` clause)
  - For example, you may want to log errors to a file – but all I/O operations require **IOException** to be caught!
- Do this by nesting a `try/catch` (and optional `finally`) sequence inside your handler

```
try {
    // processing
} catch (MyException) {
    try {
        // log error
    } catch (IOException ioe) {
        ioe.printStackTrace();
    } finally {
        // close error log file
    }
}
```

## *The throw Keyword*

---

- Can be used in a `try` block when you want to deliberately throw an exception
- The `throw` statement requires a single argument:
  - A *throwable* object
  - `throw someThrowableObject;`
- To encapsulate the condition, create a new instance of the exception class
- The flow of the execution stops immediately after the `throw` statement and the next statement is not reached
  - A `finally` clause will still be executed if present

```
throw new java.io.IOException("msg");
```

## Assertions

---

- An *assertion* is a Java statement that allows you to test your assumptions about your program
  - In a traffic simulator, you might want to assert that a speed is positive, yet less than a certain maximum
- An assertion contains a boolean expression that you believe will be true when the assertion executes – if not true, the system throws an error
- Benefits:
  - Writing assertions while programming is a quick and effective way to detect and correct bugs
  - Assertions document the inner workings of your program, enhancing maintainability

## Using Assertions

---

- Two forms:
  - `assert <boolean expression> ;`
  - `assert <boolean expression> : <value expression> ;`
- If the boolean expression is false:
  - Form 1 throws an **AssertionError** with no message
  - Form 2 throws an **AssertionError** with a message defined by evaluating the second expression
- Assertion checking is disabled by default
  - Must be enabled at java command line using the `enableassertions` switch
  - Assertions can be enabled or disabled on a package-by-package or class-by-class basis
  - `assert` statements are ignored if not enabled

## ***When to Use Assertions***

---

- Do **not** use assertions:
  - For argument checking in public methods
    - Requires a **RuntimeException**, such as **IllegalArgumentException**
  - For any work your application requires for correct operation
- Use assertions to test:
  - Internal Invariants (values that should never occur)
    - e.g., place “default: assert false” at the end of switch statements with no default
  - Control-Flow Invariants
    - e.g., place “assert false” at locations that should never be reached
  - Preconditions, Post Conditions, and Class Invariants
    - e.g., argument checking in private methods

## ***What You Have Learned***

---

- In this unit, you have learned how
  - Exceptions are used to handle error conditions
  - Exceptions are delegated
  - `try`, `catch` and `finally` are used
  - Exceptions are thrown
  - The `assert` statement is used