Exception-Handling
Java Programming
PRELUDE

Exception-handling,
Exception Types, Uncaught Exceptions, Exception Clauses

try, catch, throw, throws, finally,
Java’s Built-in Exceptions
Exception-Handling

A Java exception is an object that describes an exceptional error condition that has occurred in a piece of code at "RUN TIME"
Exception

When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
Diagrammatic Representation of Program Execution

Java Code

Java Compiler
javac

classes.zip: classes needed at runtime by Java Interpreter

Exception Classes

Java Interpreter
java

Bytecode

Error or Exception

Exception Type determined

Object of Exception Class created

Related message displayed

No Errors

Program Execution
Exceptions can be generated by:
Java run-time system:-
ERROR
Generated by the code:-
EXCEPTION
THROWABLE

- All Exception types are subclasses of the built-in class THROWABLE.

- THROWABLE is at the top of the Exception class Hierarchy.
Exception Types

Throwable

Exception

Run-time Exception

ClassNotFound

Error

Linkage Error

Stack overflow

ArrayIndexOutOfBoundsException
Uncaught Exceptions

class UncaughtEx{
    public static void main(String args[]) {
        int d = 0;
        int a = 42/d;
    }
}

Output:
java.lang.ArithmeticException: / by zero
at Exc0.main(UncaughtEx.java:4)
Five keywords
Java exception-handling

- try
- catch
- throw
- throws
- finally
Exception-Handling ...

- Program statements to be monitored for exceptions are contained within a *try* block.

- Your code can catch this exception using *catch* and handle it in some rational manner.
- Use the keyword `throw` to throw an exception.

- Any exception that is thrown out of a method `throws` clause.

- Any code that absolutely must be executed is put in a `finally` block.
class HandledException{
    public static void main(String args[])
    {
        int d, a;
        try {
            d = 0;
            a = 42 /d;
        }
        catch(ArithmeticException e)
        {
            System.out.println(“Division by 0”);
        }
    }
}
Output: Division by 0
Using *try* and *catch*

Advantages of handling exceptions:

👉 It allows the programmer to fix the error

👉 Prevents the program from automatically terminating
try-catch Control Flow

- code before try
- try block
- exception occurs
- catch block
- code after try
try-catch Control Flow

code before try

exception occurs

try block

catch block

no exceptions occur

code after try
Multiple *catch* clauses

- One block of code causes multiple Exception.
- Two or more *catch* clauses.

- Exception subclass must come before any of their super classes.
- Unreachable code.
public static void main(String args[]) {
    try {
        int a = args.length;
        int b = 42 / a;
        int c[] = {1}; c[42] = 99;
    }
    catch (ArithmeticException e) {
        System.out.println(“Divide by 0: “ + e);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.out.println(“Array index oob: “ + e);
    }
    System.out.println(“After try/catch block”);
}
Nested *try* Statements

- A *try* and its *catch* can be nested inside the block of another *try*.

- It executes until one of the *catch* statements succeeds or Java run-time system handle the exception.
try{ int a=args.length;
    int b=42/a;
    try{int c[ ]={1},c[40]=99;}
    catch(ArrayIndexOutOfBoundsException e)
    {System.out.println(e); }
}

catch(ArithmeticException e)
{ }
The *finally* Clause

*finally* creates a block of code that will be executed (whether or not an exception is thrown)
class FinallyDemo {
    int [ ] num1= {12, 16, 10, 8, -1, 6};
    int [ ] num2 = { 1, 5, 35, 20, 1, 13};

    public static void main(String [ ] args) {
        FinallyDemo f = new FinallyDemo( );
        f.readNums(f.num1);
        f.readNums(f.num2);}
}
void readNums(int[] array) {
    int count = 0, last = 0;
    try {
        while (count < array.length) {
            last = array[count++];
            if (last == -1) return;
        }
    } finally {
        System.out.println("Last" + last);
    }
}
**try-catch Control Flow**

- **code before try**
- **exception occurs**
- **try block**
- **catch block**
- **finally block (if it exists)**
- **code after try**
The *throw* clause

- Throw an exception explicitly.

\[ \textit{throw} \text{ ThrowableInstance} \]

**Throwable object:**

- Using a parameter into a *catch* clause
- Creating one with the *new* operator.
```java
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullpointerException("demo");
        } catch (NullPointerException e) {
            System.out.println("demoproc.");
            throw e;
        }
    }
    // Output: demoproc
```
Continued…

```java
public static void main(String[] args) {
    try {
        demoproc();
    } catch (NullPointerException e) {
        System.out.println("Recaught" + e);
    }
}
```

//Recaught: java.lang.NullPointerException: demo
Example

```
Method A
try {
    B();
}
catch (Exception e) {
    output.println("A");
}

Method B
try {
    C();
}
catch (Exception e) {
    output.println("B");
}

Method C
D();

Method D
if (cond) {
    throw new Exception();
}
```

Call Sequence

Method A → Method B → Method C → Method D

Stack Trace

A → B → C → D
The *throws* clause

- A *throws*: is used to throw a Exception that is not handled.

- **Error** and **RuntimeException** or any of their subclasses don’t use throws.
The *throws* clause -continued

- *Type* `method-name (parameter-list) throws exception-list`

```java
{ // body of method }
```
class ThrowsDemo {
    static void throwProc( ) throws IllegalAccessException {
        throw new IllegalAccessException("demo");
    }

    public static void main (String args[]) {
        try {
            throwProc( );
        }
        catch(IllegalArgumentException e) {
            System.out.println("Caught " + e);
        }
    }
}
Java’s Built-in Exceptions

- Java defines several exception classes inside the standard package `java.lang`
  - `RuntimeException` or `Error`
Unchecked Exceptions

Unchecked Exception = Runtime Exceptions/ERROR

Example:
- `NumberFormatException`
- `IllegalArgumentException`
- `OutOfMemoryError`
Checked Exceptions

Checked Exception = checked at compile time

⚠️ These errors are due to external circumstances that the programmer cannot prevent

 пример: IOException
Java’s Built-in Exceptions ...

The following are Java’s Checked Exceptions:

- `ClassNotFoundException`
- `CloneNotSupportedException`
- `IllegalAccessException`
- `InstantiationException`
- `InterruptedException`
- `NoSuchFieldException`
- `NoSuchMethodException`
- `UnsupportedOperationException`
PRELUDE

Garbage collection, User defined Exceptions, Assertions, chained Exceptions
The Utility of Backtracking

Backtracking is the process by which the stack frame is unwound in the presence of an unhandled exception.

የሚ_objs created on the stack are discarded to the GC.
Garbage Collection

- When no references to an object exists that object is assumed to be not needed.
- Java’s garbage collector (GC) offers each method a type of destructor called ‘finalizer()’.
- It is called with **neither timing nor order** guaranteed.
- To ensure that the resources are freed finalize() method.

```java
****System.gc();
protected void finalize()
{
    //finalization code
}
```
Creating Your Own Exception Classes

- All user-created exceptions – subclass of Exception
- All methods inherited Throwable.
Demo of ExcepDemo.java

class YourException extends Exception
{
    private int detail;
    YourException(int a)
    {
        detail = a;
    }
    public String toString()
    {
        return "YourException[" + detail + "]";
    }
}
```java
class ExcepDemo {
    static void compute(int a) throws YourException {
        if (a > 10) throw new YourException(a);
        System.out.println("Normal Exit");
    }
}
```
public static void main(String args[]){
    try {
        compute(1); compute(20);
    } catch (YourException e) {
        System.out.println("Caught" + e);
    }
}
Output:

Called compute(1)
Normal exit
Called compute(20)
Caught YourException[20]
Assertions

- Assertions are conditions that should be true at a particular point in a method.

- Assertions can be validated using the `assert` statement
Assertions

- `assert` statement

Evaluates -true or false

```java
assert exp;
```

AssertionError if exp false

```java
assert exp1 :exp2;
```

exp2 is error message
By default, assertions are disabled

Assertions can be enabled with the 

`-ea` command-line option
assert
(number>=0&&number<=10):
"bad number"+ number

If you enter a number 50

Exception in thread "main"
java.lang.AssertionError:
bad number:50
Chained Exceptions

Associate one exception with another.

Second exception describes the cause of the first exception.

`initCause(Throwable causeExc)`  
`getCause()`
static void demo(){
    NullPointerException e=new NullPointerException(“Top”);
    e.initCause(new ArithmeticException(“cause”));
    throw e;
}

e.getCause();

NullPointerException, Original Cause ArithmeticException.
When to Use Exceptions

☞ Deal with the exception `try` and `catch`.

☞ Pass the exception up the calling chain by adding `throws` clause.

☞ Both by catching the using `catch` and then explicitly rethrowing it using `throw`.
When Not to Use Exceptions

- When exception is something that is expected and could be avoided easily with a simple expression.

- Exceptions take up lot of processing time. A simple test or series of tests will run much faster than exception-handling.
Java exception system was designed to warn users for the possibility of their occurrence. Ignoring them could have resulted into fatal errors.

Even worse, adding `throws` to your methods to avoid exceptions means that the users of your methods will have to deal with them - method made more difficult to use.
Why Runtime Exceptions are Not Checked

Many of the operations and constructs of the Java language can result in runtime exceptions. **The information available to a Java compiler**

The level of analysis the compiler performs, are usually not sufficient to establish that such runtime exceptions cannot occur.