# JAVA Exception

## Objective/Outline

- Objective: Study how to handle exceptions in java

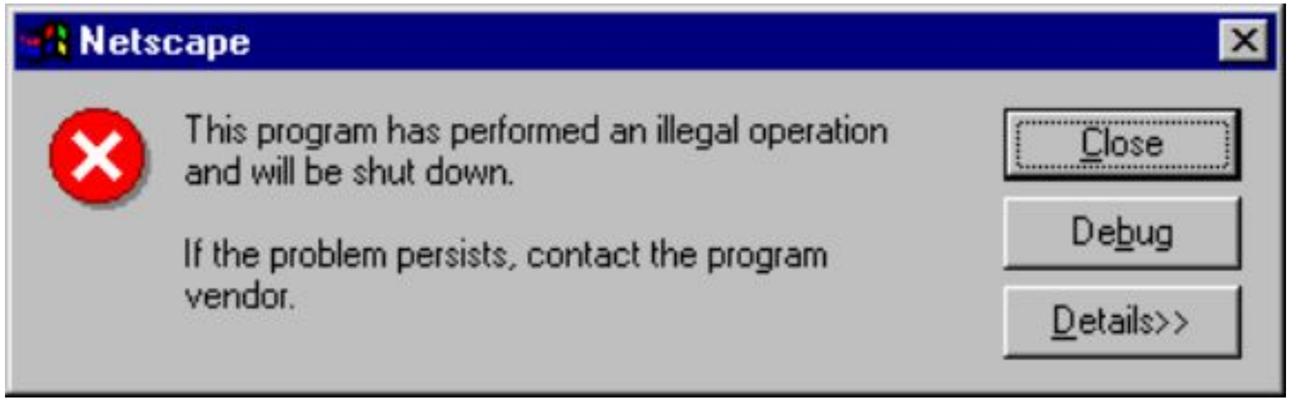
- Outline:
  - Introduction
  - Java exception classes (Checked vs unchecked exceptions)
  - Dealing with exceptions
    - Throwing exceptions
    - Catching exceptions

## Introduction

- Causes of errors
  - User input errors:
    typos, malformed URL, wrong file name, wrong info in file…
  - Hardware errors:
    Disk full, printer out of paper or down, web page unavailable…
  - Code errors:
    invalid array index, bad cast, read past end of file, pop empty stack, null object reference…

# Introduction

- Goals of error handling
  - Don't want:



Netscape

This program has performed an illegal operation and will be shut down.

If the problem persists, contact the program vendor.

  - Want:
    - Return to a safe state and enable user to execute other commands
    - Allow user to save work and terminate program gracefully.

# Introduction

- Java exception handling mechanism:

  Every method is allowed to have two exit paths

  - No errors occur
    - Method exits in the normal way
    - Returns a value
    - Control passed to the calling code.

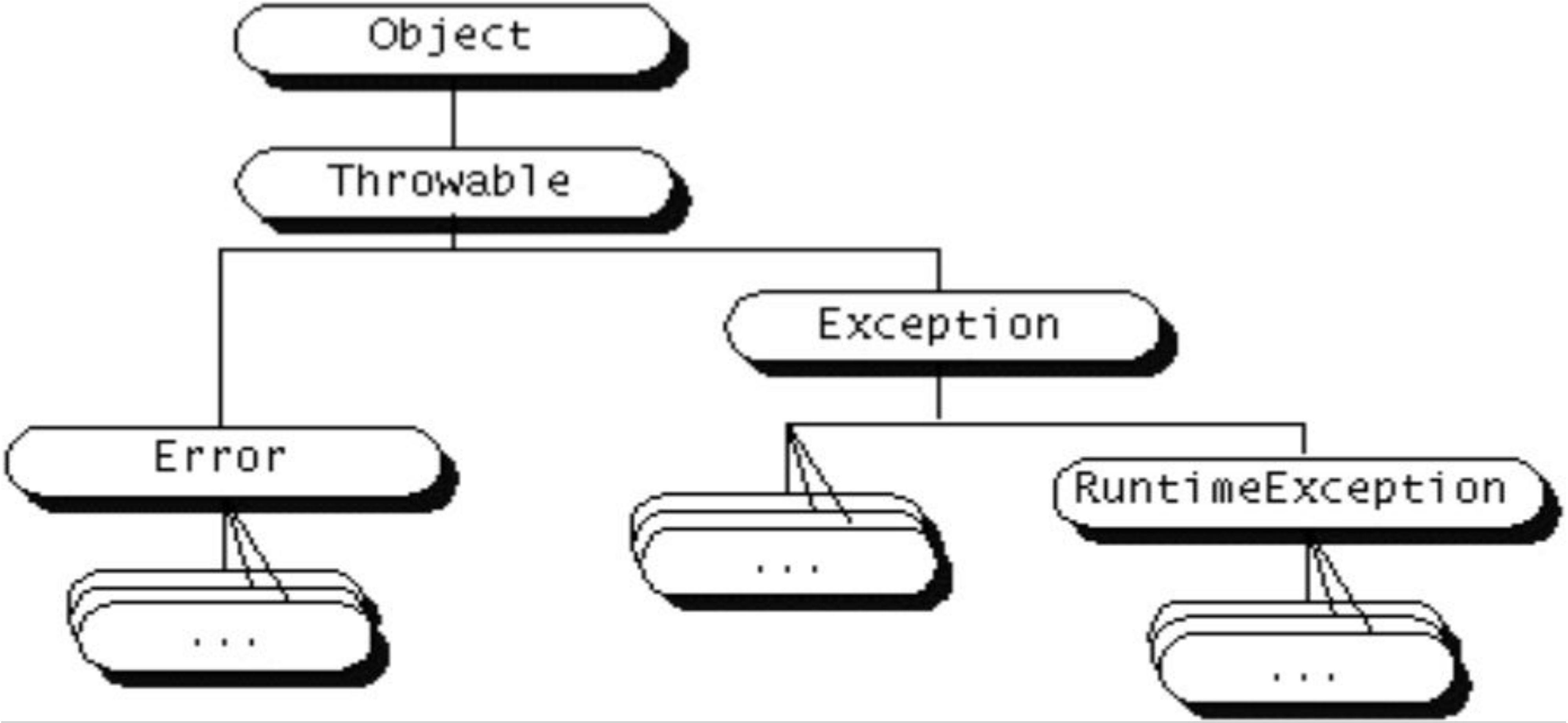
  - If errors occur
    - Method exits via an alternative exit path
    - Throws an object that encapsulates the error information
    - Control passed to exception mechanism that searches for an appropriate exception handler to deal with the error condition

## Outline

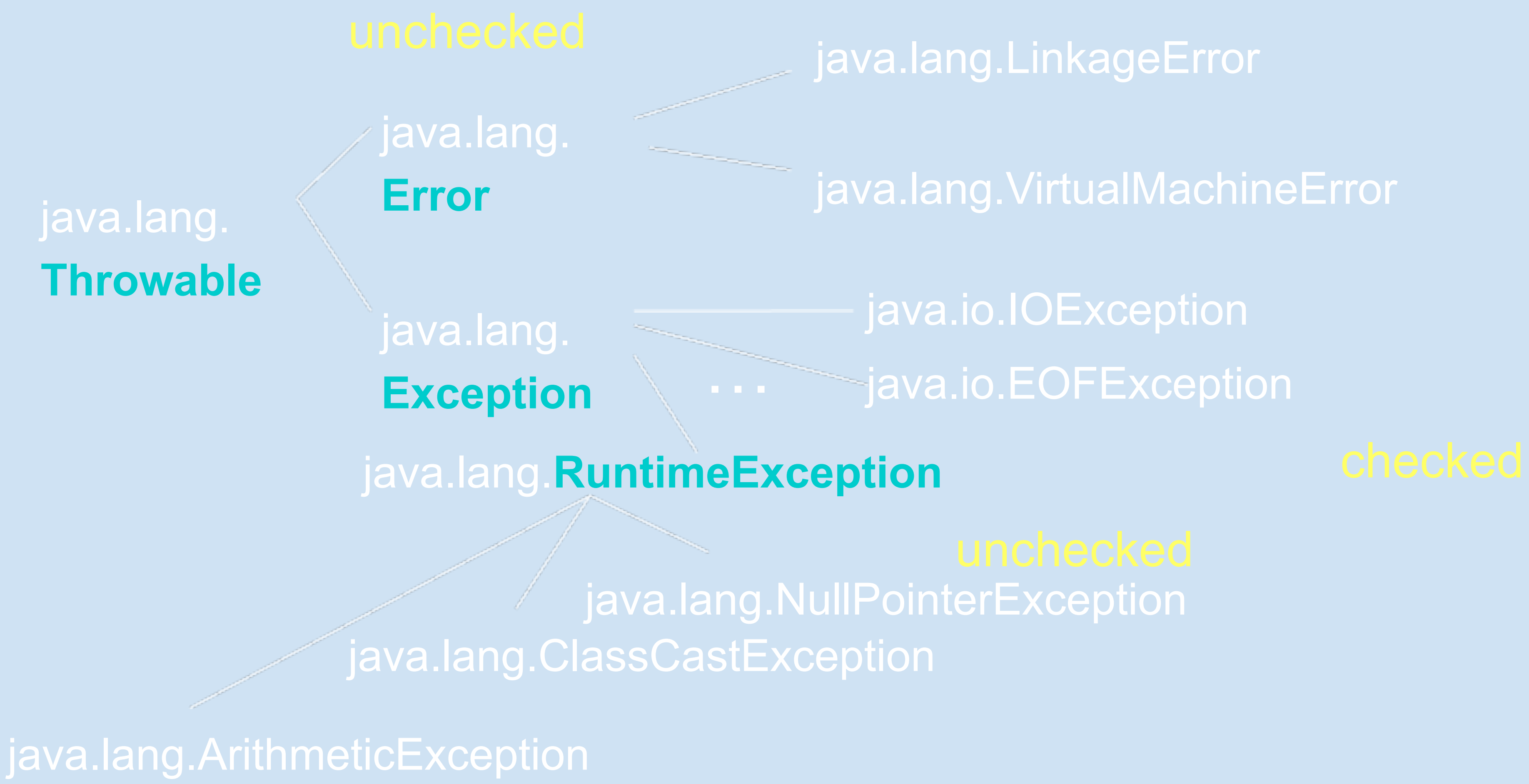
- Introduction
- Java exception classes
- Dealing with exceptions
- Throwing exceptions
- Catching exceptions

# Java Exception Classes

- Java has many classes for error handling. They are called exception classes



## Java Exception Classes



## Unchecked exceptions

- Error: For internal errors in JVM.
- RuntimeException: Logical errors in program (C++ logical-error).
  - Bad cast
  - Out-of-bound array access
  - Null reference access
- Those two exceptions are **unchecked**
  - JVM internal errors beyond your control
  - You should not allow logical errors at the first place

## Checked exceptions

- All other exceptions (C++ runtime_error) are checked, i.e. you have to explicitly handle them. Otherwise compiler errors results in.
    - Trying to read pass end of file
    - Open a malformed URL
    - ...

AclNotFoundException, ActivationException,

AWTException, BadLocationException,

ClassNotFoundException, CloneNotSupportedException,

DataFormatException, ExpandVetoException,

GeneralSecurityException, IllegalAccessException,

InstantiationException, InterruptedException,

IntrospectionException, InvocationTargetException,

IOException, LastOwnerException,

...

## Exceptions

- Typically, what methods does an exception class have?
    - Check IOException

## Java Exception Classes

- You can define new Exception classes.

```
class FileFormatException extends IOException
{ // default constructor
public FileFormatException() {}
//constructor contains a detailed message
public FileFormatException(String message)
{ super( message );
}
```

  - New Exception class must be subclass of Throwable
  - Most programs throw and catch objects that derive from the Exception class

## Outline

- Introduction
- Java exception classes
- Dealing with exceptions
- Throwing exceptions
- Catching exceptions

## Dealing with Exceptions

- Need to consider exceptions when writing each method
  - Identifying possible exceptions
    - Check each line of code inside the method
      - If a method from someone else is called, check API to see if it throws exceptions
      - If some other method you wrote is called, also check to see if it throws exceptions.
  - Dealing with exceptions
    - If **checked** exceptions might be thrown at any point inside the method, you need to deal with it
      - Catching exceptions: Handle an exception in the current method.
      - Throwing exceptions: Don’t know how to handle an exception in the current method and need the caller method to deal with it.

## Throwing Exceptions

- Throw an exception generated by method call:

```
public void readData(BufferedReader in)throws IOException
{
String s = in.readLine();
StringTokenizer t = new StringTokenizer(s, "|");
name = t.nextToken();
salary = Double.parseDouble(t.nextToken());
int y = Integer.parseInt(t.nextToken());
int m = Integer.parseInt(t.nextToken());
int d = Integer.parseInt(t.nextToken());
GregorianCalendar calendar
= new GregorianCalendar(y, m - 1, d);
// GregorianCalendar uses 0 = January
hireDay = calendar.getTime();
}//DataFileTest.java from Topic 5
```

# Throwing Exceptions

- The method **readLine** of **BufferedReader** throws an IOExpcetion, a checked exception
  - We do not deal with this exception in the current method. So we state that the readData method might throw **IOException**.

- If you simply ignore it, compiler error results in. Try this.

- The nextToken method of StringTokenizer might throw NoSuchElementException. But it is not checked, so we don’t have to deal with it.

## Throwing Exceptions

- Notes:
  - Can throw multiple types of exceptions

```
    public void readData(BufferedReader in)
throws IOException, EOFException
```

  - Overriding method in subclass cannot throw more exceptions than corresponding method in superclass
    - If method in superclass does not throw any exceptions, overriding method in subclass cannot either

## Outline

- Introduction
- Java exception classes
- Dealing with exceptions
- Throwing exceptions
- <u>Catching exceptions</u>

## Catching Exceptions

- Catch exceptions with **try/catch** block

```
try
{ code
more code
}
catch( ExceptionType e)
{ handler for this exception
}
```

  - If a statement in the **try** block throws an exception
    - The remaining statements in the block are skipped
    - Handler code inside **catch** block executed.

  - If no exceptions are thrown by codes in the **try** block, the **catch** block is skipped.

## Dealing with Exceptions

- Example:

```
try {
    average = total/count;
    System.out.println("Average is " + average); }
catch (ArithmeticException e) {
    System.out.println("Oops: "+ e);
    average = -1;}
```

- If **count** is 0, this code will print out something like "**Oops: division by zero**".

## Catching Exceptions

```
public static void main(String[] args)
    { try
    { BufferedReader in = new BufferedReader(
    new FileReader(args[0]));
    Employee[] newStaff = readData(in);
    …
    }
    catch(IOException exception)
    { exception.printStackTrace();
    }} //ExceptionTest.java
```

- This code will exit right away with an error message if something goes wrong in **readData** or in the constructor of **FilerReader** (an **IOException** will be thrown)

# Catching Multiple Exceptions

- Can have multiple catchers for multiple types of exceptions:

```
public static void main(String[] args)
{ try
{ BufferedReader in = new BufferedReader(
new FileReader(args[0]));
Employee[] e = readData(in);
… }
catch(IOException e1)
{ exception.printStackTrace(); }
catch(ArrayIndexOutOfBoundsException e2)
{ System.out.print("No file name provided " );
System.exit(1); }
} // ExceptionTest2.java
```

Might throw ArrayIndexOutOfBoundsExpection

**What if GeneralSecurityException occurs in the try block?**

## Dealing with Exceptions

- Note that the following will produce a compiling error. Why?

```
try {…}
catch (Exception e3) {…}
catch (ArithmeticException e1){…}
catch (IOException e2) {…}
```

# Catching Exceptions

- Catchers can also re-throw an exception or throw exception that is different from the exception caught.

```
graphics g = image.getGraphics();
try { …}
catch (MalformedURLException e)
{ g.dispose();
throw e;
}
```

We wish to dispose the graphics object g, but we don’t know how to deal with the exception.

How to create a new exception and throw it?

# The finally clause

```
try
{ code
more code}
catch( ExceptionType e)
{ handler for this exception }
finally
{ .. }
```

- The **finally** block is executed regardless whether exceptions are thrown in the **try** block.
- Useful in situations where resources must be released no matter what happened

# The finally clause

- A caution about the **finally** clause:
  - Codes in the **finally** block are executed even there are **return** statements in the **try** block

```
public static int f(int n)
{ try
{ return n* n;
}
finally
{ if ( n==2) return 0;
}
}
```

f(2) return 0 instead of 4!

## Dealing with Exceptions

- Search for handler: Steps:
  - Tries to find a handler in the **Catch** block for the current exception in the current method. Considers a match if the thrown object can legally be assigned to the exception handler's argument.
  - If not found, move to the caller of this method
  - If not there, go another level upward, and so on.
  - If no handler found, program terminates.