

l) An example of a built-in exception

Although it is unlikely, the `readLine( )` method can fail, Java requires that you at least acknowledge the possibility of failure. This is done by adding the line

**import** `Java.io.IOException;` in the import section of the program,  
and adding the clause

**throws** `IOException` to the declaration of **public static void** `main(String[ ] args)`  
to make

**public static void** `main(String[ ] args)` **throws** `IOException`

This is what will produce the built-in error message in to the Java program.

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
public class ex1
{
    public static void main (String[ ] args) throws IOException
    {
        InputStreamReader inStream;
        BufferedReader keyboard;
        inStream = new InputStreamReader(System.in);
        keyboard = new BufferedReader(inStream);
        String input;
        int num;
        System.out.println("Type in an integer then <Enter>");
        input = keyboard.readLine( );
        num = Integer.parseInt(input);
        System.out.println("Your input now as an integer is: " + num);
    }
}
```

\*Since `main` throws the exception and there is no `try` then if an error occurs it is thrown back to the JVM and a prewritten error message is output and the program would then terminate. This is what usually happens with checked exceptions and unchecked ones that have not `try/catch`.

### H) Designing Your Own Exception Types

You can design your own exception types by extending `RuntimeException` or one of its subclasses

ex.1)

```
if (amount > balance)
{
    throw new InsufficientFundsException(
        "withdrawal of " + amount + " exceeds balance of " + balance);
}
```

Note: You are making it an unchecked exception  
(programmer could have avoided it by calling `getBalance` first)

**finally** summary:

- Executed when try block is exited in any of three ways:
  1. After last statement of try block
  2. After last statement of catch clause, if this try block caught an exception
  3. When an exception was thrown in try block and not caught
- Recommendation: don't mix catch and finally clauses in same try block

```
try
{
    statement
    statement
    ...
}
finally
{
    statement
    statement
    ...
}
```

ex.2)

```
FileReader reader = new FileReader(filename);
```

```
try
{
    readData(reader);
}
finally
{
    reader.close( );
}
```

Purpose:

To ensure that the statements in the finally clause are executed whether or not the statements in the try block throw an exception.

### G) The **finally** Clause

An exception terminates current method. Danger: Can skip over essential code  
ex. 1)

```
reader = new FileReader(filename);  
Scanner in = new Scanner(reader);  
readData(in);  
reader.close( ); // May never get here
```

Must execute reader.close( ) even if exception happens

Use finally clause for code that must be executed "no matter what":

```
FileReader reader = new FileReader(filename);  
try  
{  
    Scanner in = new Scanner(reader);  
    readData(in);  
}  
finally  
{  
    reader.close( ); // if an exception occurs, finally clause is also  
                    // executed before exception is passed to its handler  
}
```

\*Note: The **catch** and **finally** are both optional. You can have one or both in after try block

F) General **try** Block:

```
try
{
    statement
    statement
    ...
}
catch (ExceptionClass exceptionObject)
{
    statement
    statement
    ...
}
catch (ExceptionClass exceptionObject)
{
    statement
    statement
    ...
}
...
```

ex.1)

```
try
{
    System.out.println("How old are you?");
    int age = in.nextInt();
    System.out.println("Next year, you'll be " + (age + 1));
}
catch (InputMismatchException exception)
{
    exception.printStackTrace();
}
```

Purpose:

To execute one or more statements that may generate exceptions. If an exception occurs and it matches one of the catch clauses, execute the first one that matches. If no exception occurs, or an exception is thrown that doesn't match any catch clause, then skip the catch clauses.

## E) Catching Exceptions

- Install an exception handler with try/catch statement
- **try** block contains statements that may cause an exception
- **catch** clause contains handler for an exception type
  - ex.1)
  - **try**
  - {
  - String filename = . . . ;
  - FileReader reader = **new** FileReader(filename);
  - Scanner in = **new** Scanner(reader);
  - String input = in.next( );
  - **int** value = Integer.parseInt(input);
  - . . .
  - }
  - **catch** (IOException exception)
  - {
  - exception.printStackTrace();
  - }
  - **catch** (NumberFormatException exception)
  - {
  - System.out.println("Input was not a number");
  - }

## Catching Exceptions Summary:

- Statements in **try** block are executed
- If no exceptions occur, **catch** clauses are skipped
- If exception of matching type occurs, execution jumps to **catch** clause
- If exception of another type occurs, it is thrown until it is caught by another try block
- catch (IOException exception) block
  - exception contains reference to the exception object that was thrown
  - catch clause can analyze object to find out more details
  - exception.printStackTrace(): printout of chain of method calls that lead to exception

### D) Checked and Unchecked Exceptions continued

Two choices:

1. Handle the exception using a try/catch
2. Tell compiler that you want the method to be terminated when the exception occurs due to no try/catch

- Use **throws** specifier so method can throw a checked exception

```
public void read(String filename) throws FileNotFoundException
{
    FileReader reader = new FileReader(filename);
    Scanner in = new Scanner(reader);
    ...
}
```

- For multiple exceptions:

```
public void read(String filename)
throws IOException, ClassNotFoundException
```

Keep in mind inheritance hierarchy : If method can throw an IOException and FileNotFoundException, only use IOException if not going to use both.

Note: It is better to declare an exception using throws and let the JVM worry about it than to handle it incompetently in a try/catch.

### C) Two types of Exceptions: Checked and Unchecked

Checked (means compiler-checked)

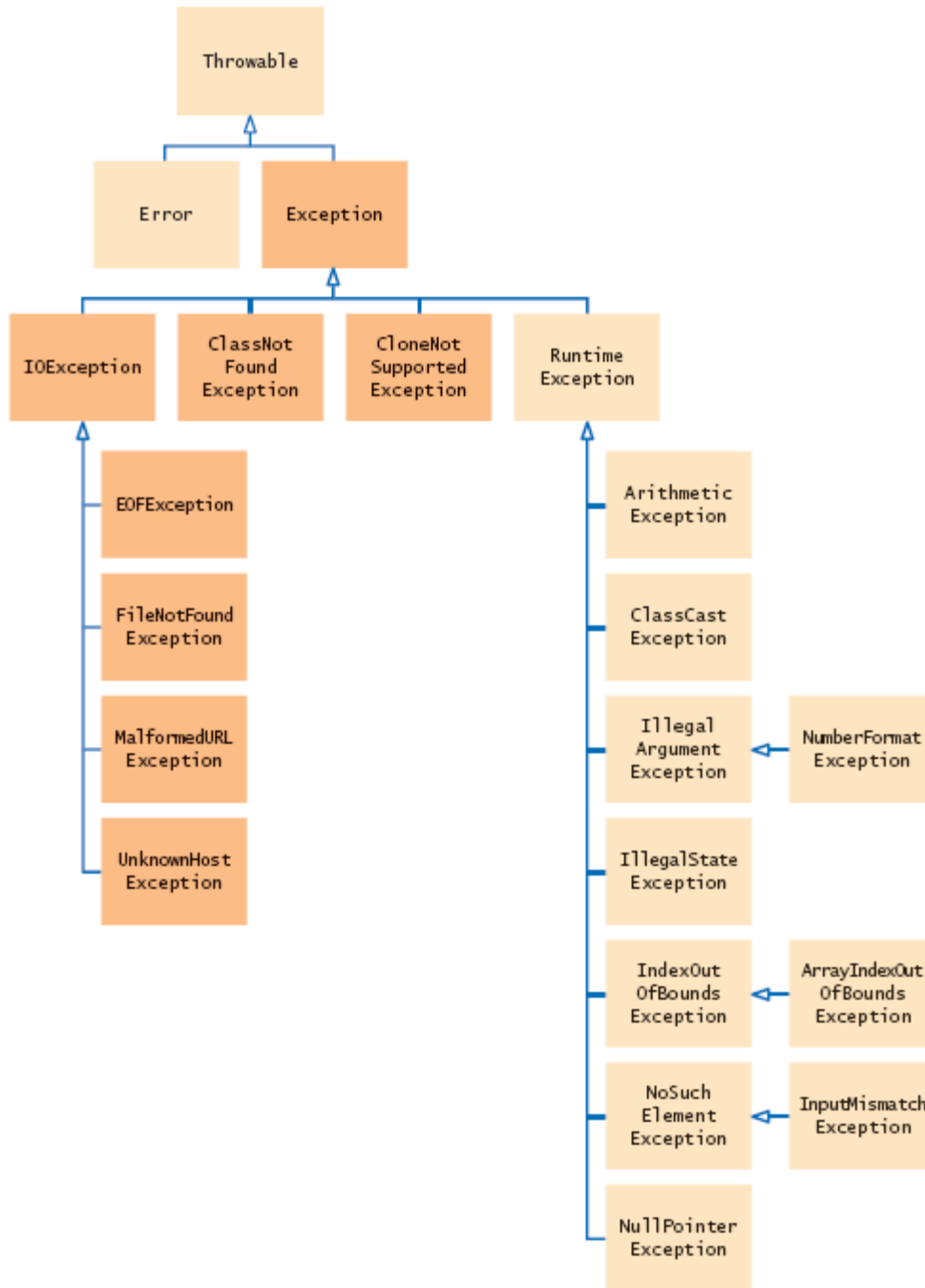
- Due to external circumstances that the programmer cannot prevent
- The compiler checks that you don't ignore them
- Majority occur when dealing with input and output ex.) IOException
- when using `readLine( )` from `BufferedReader` (or similar) the computer forces a **throws** clause to be included *or* a **try/catch** block to be written  
ex.1) **public static void main** (String[ ] args) **throws** IOException

Unchecked: (compiler does not care about these)

- Extend the class `RuntimeException` (or `Error`)
- They are the programmer's fault
- Examples of runtime exceptions:
  - NumberFormatException
  - IllegalArgumentException
  - NullPointerException
- Example of error:
  - OutOfMemoryError

\*\*Any RunTimeExceptions do not have to be declared using **throws** nor wrapped in a try/catch (but you can like the example on page 2)

Heirachy chart of the Throwable/Exception classes  
(Note: ones below inherit from the ones above.)



**Figure 1** The Hierarchy of Exception Classes

Note: Throwable has two methods that are inherited by all subsequent classes:

getMessage( ), and printStackTrace( )

```
ex.1) public class TestExceptions
{
    public static void main(String [ ] args)
    {
        String = "yes"; // then try "sure", then try "no"
        try
        {
            System.out.print("start try ");
            doRiskyMethod(test);
            System.out.print("end try ");
        }
        catch (ReallyScaryException se) // inherits from ScaryException
        {
            System.out.print("really scary exception ");
        }
        catch (ScaryException se)
        {
            System.out.print("scary exception ");
        }
        finally
        {
            System.out.print("finally ");
        }
        System.out.print("end main ");
    }
    public static void doRiskyMethod(String test)
        throws ScaryException, ReallyScaryException
    {
        System.out.print("start risky ");
        if ("yes".equals(test))
            throw new ScaryException( );
        if ("sure".equals(test))
            throw new ReallyScaryException( );
        System.out.println("end risky ");
        return;
    }
}
public class ScaryException extends Exception { }
public class ReallyScaryException extends Exception { }
```

output for String = "yes": start try start risky scary exception finally end main

output for String = "sure": start try start risky really scary exception finally end main

output for String = "no": start try start risky end risky end try finally end main

## Exception Handling

### I. Error Handling

A) Traditional approach: Method returns error code. Two problems:

1) Called method may not be able to do anything about failure so you let the program fail

2) let its caller method worry about it thus many method calls would need to be checked so instead of programming for success:

```
x.doSomething( )
```

you would always be programming every method call for failure:

```
if (!x.doSomething( ))
```

```
return false;
```

B) Throwing Exceptions was created to solve the above problems:

1) Can't be overlooked

2) Sent directly to an exception handler – not just caller of failed method

Throw an exception object to signal an exceptional condition

ex. 1) // IllegalArgumentException: illegal parameter value

```
if (amount > balance)
```

```
{
```

```
    IllegalArgumentException exception =
```

```
        new IllegalArgumentException("Amount exceeds balance");
```

```
    throw exception;
```

```
}
```

Shortcut: There is no need to store exception object in a variable:

```
throw new IllegalArgumentException("Amount exceeds balance");
```

When an exception is thrown, the method terminates immediately (just like a return) and execution continues with an exception handler.