I/O Stream, Reader, Writer, BufferedReader & BufferedWriter

- I/O in Java is built on *streams*. A stream means an unbroken flow of data (which could be bytes, characters, objects, etc.)
- Input stream connects a data source (e.g. a file, a string, an array, or network connection) to a Java program.
- Output stream connects a Java program to a data sink (e.g. monitor, printer, file, or network connection).
- Different stream classes read/write particular sources of data, e.g.
  `java.io.FileInputStream` reads data from a file
  `java.lang.System.in` reads data from keyboard

- An `InputStreamReader` is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified `charset`.
- The `charset` that it uses may be specified by name or may be given explicitly, or the platform's default `charset` may be accepted.

- An `OutputStreamWriter` is a bridge from character streams to byte streams.
- Characters written to it are encoded into bytes using a specified `charset`.

- A `Reader` is an abstract class for reading character streams.
- Similarly a `Writer` is an abstract class for writing to character streams.
- The `read` method of the `Reader` class reads a single character.
- The `write` method of the `Writer` class writes a single character.

- Reading or writing a single byte or character at a time is often inefficient.
- `BufferedReader` provides buffering support to a `Reader`. It is advisable to wrap a `BufferedReader` around any `Reader` whose read operations may be costly.

  Example codes:

```
BufferedReader in = new BufferedReader(
                            new InputStreamReader(System.in));
String line = in.readLine();
```

- Similarly, it is advisable to wrap a `BufferedWriter` around any `Writer` to improve I/O efficiency.

Basic Text I/O

A convenient way to process text-based input is to use a **`Scanner`**.

Constructor Summary
`Scanner`(`File` `source`) Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner`(`File` `source`, `String` `charsetName`) Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner`(`InputStream` `source`) Constructs a new `Scanner` that produces values scanned from the specified input stream.
`Scanner`(`InputStream` `source`, `String` `charsetName`) Constructs a new `Scanner` that produces values scanned from the specified input stream.
`Scanner`(`Readable` `source`) Constructs a new `Scanner` that produces values scanned from the specified source.
`Scanner`(`ReadableByteChannel` `source`) Constructs a new `Scanner` that produces values scanned from the specified channel.
`Scanner`(`ReadableByteChannel` `source`, `String` `charsetName`) Constructs a new `Scanner` that produces values scanned from the specified channel.
`Scanner`(`String` `source`) Constructs a new `Scanner` that produces values scanned from the specified string.

You can create a `Scanner` from a file, an input stream (e.g. `System.in`), or even a `String` object.

A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace.

The resulting tokens may then be converted into values of different types using the various `next` methods, e.g. `next()`, `nextLine()`, `nextInt()`, `nextDouble()`, etc.

To test if there is another token, use the methods `hasNext()`, `hasNextLine()`, `hasNextInt()`, etc.

For example, this code allows a user to read a number from `System.in`:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

As another example, this code reads in `long` numbers from a text file `myNumbers.txt`

```
Scanner sc = new Scanner(new File("myNumbers.txt"));
while (sc.hasNextLong())
{
    long aLong = sc.nextLong();
}
```

The scanner can also use delimiters other than whitespace. This example reads several items in from a string:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input);
s.useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

prints the following output:

```
1
2
red
blue
```

The `next()` method reads a word (token) at a time.

The `nextLine()` method reads a line at a time. The input line is read into a string, and your program can then process the string.

Example: process a file with population data lines like this

```
China  1330044605
India  1147995898
United States 303824646
```

- First read each input line into a string
- Then use the `isDigit` and `isWhitespace` methods to find out where the name ends and the number starts. Then extract the country name and population.

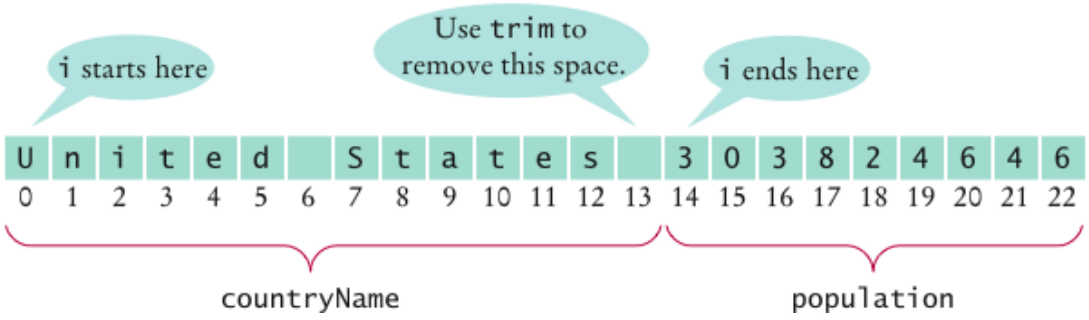
```
String line = sc.nextLine();
int i = 0;
while (!Character.isDigit(line.charAt(i)))
   i++;

String countryName = line.substring(0, i);
//substring: start index = 0, end index = i (excluded)

String population = line.substring(i);  //start index = i
```

- Use the `trim` method to remove spaces at the end of the country name:

```
countryName = countryName.trim();
```



- To convert the population string to a number, first trim it, then call the method `Integer.parseInt`

```
int populationValue = Integer.parseInt(population.trim());
```

Alternative approach to process text input in the above format:

```
// preconditions:
// last token is the population
// country name consists of the first n-1 tokens

String line = sc.nextLine();
String[] tokens = line.split("\\s");

String countryName = tokens[0];
for (int i = 1; i < tokens.length – 1; i++)
   countryName = countryName + " " + tokens[i];

String population = tokens[tokens.length-1];
```

Reading numbers

- `nextInt` and `nextDouble` methods consume white space (that precedes the number) and the next number.

```
double value = sc.nextDouble();
```

- If there is no number in the input, then a `InputMismatchException` occurs

- To avoid exceptions, use the `hasNextDouble` and `hasNextInt` methods to screen the input:

```
if (sc.hasNextDouble())
{
   double value = sc.nextDouble();
   . . .
}
```

The `nextInt` and `nextDouble` methods do not consume the white space that follows a number.

- Example: file contains student IDs and names in this format:

```
1729
Harry Morgan
1730
Diana Lin
```

- Read the file with these instructions:

```
while (sc.hasNextInt())
{
   int studentID = sc.nextInt();
   String name = sc.nextLine();   //error
   Process the student ID and name
}
```

- After the first call to `nextInt`, the input contains

- The call to `nextLine` reads an empty string! The remedy is to add a call to `nextLine` after reading the ID:

```
int studentID = sc.nextInt();
sc.nextLine(); // Consume the newline

String name = sc.nextLine();
```

- To read one character at a time, set the delimiter pattern to the empty string:

```
Scanner sc = new Scanner(. . .);
sc.useDelimiter("");
```

- Now each call to `next` returns a string consisting of a single character
- To process the characters:

```
while (sc.hasNext())
{
   char ch = sc.next().charAt(0);
   Process ch;
}
```

Alternatively, you can use a `read` method of `FileReader or Reader` to read 1 character at a time.

To write to a file, you can use a `PrintWriter` object.

```
PrintWriter out = new PrintWriter("output.txt");
out.println(29.95);
out.println("Hello, World!");
out.close(); //you must close the file when you are done
```

- If the file already exists, it is emptied before new data are written into it.
- If the file doesn't exist, an empty file is created.

Example:

- Reads all lines of a file and sends them to the output file, preceded by line numbers
- Sample input file:

```
Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```

- Program produces the output file:

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```

- Program can be used for numbering Java source files

```
 1  import java.io.File;
 2  import java.io.FileNotFoundException;
 3  import java.io.PrintWriter;
 4  import java.util.Scanner;
 5
 6  /**
 7     This program applies line numbers to a file.
 8  */
 9  public class LineNumberer
10  {
11     public static void main(String[] args)
                        throws FileNotFoundException
12     {
13        // Prompt for the input and output file names
```

```
14
15          Scanner console = new Scanner(System.in);
16          System.out.print("Input file: ");
17          String inputFileName = console.next();
18          System.out.print("Output file: ");
19          String outputFileName = console.next();
20
21          // Construct Scanner & PrintWriter objects
22
23          File inputFile = new File(inputFileName);
24          Scanner in = new Scanner(inputFile);
25          PrintWriter out = new PrintWriter(outputFileName);
26          int lineNumber = 1;
27
28          // Read the input and write the output
29
30          while (in.hasNextLine())
31          {
32             String line = in.nextLine();
33             out.println("/* " + lineNumber + " */ " + line);
34             lineNumber++;
35          }
36
37          in.close();
38          out.close();  //must close the file explicitly
39       }
40    }
```

If you want to append new text to an existing file, you need to use a `FileWriter` and `BufferedWriter`.

Sample codes:

```
try
{
   FileWriter fstream = new FileWriter(“myData.txt”, true);
   BufferedWriter out = new BufferedWriter(fstream);
   out.write(“Hello Java”);
   out.close();
}
catch (IOException e)
{
   System.err.println(“Error: ” + e.getMessage());
}
```

Exception Handling

When a method detects a problematic situation, what should it do?

- Include additional code in the method to handle the situation
  - drawback: how to handle failures is context dependent, the programmer cannot foresee all possible scenarios
- The method returns an indicator whether it succeeded or failed, and leave the problem to the caller to handle
  - drawback: the caller may forget to check the return value, or may not be able to do anything about the failure

The exception-handling mechanism of Java is designed to solve this problem

- Exceptions cannot be overlooked.
- Exceptions are handled by a competent handler – not just the caller of the failed method.

Checked exceptions

- When you call a method that throws a checked exception, you must tell the compiler what you are going to do about the exception if it is ever thrown.
- Checked exceptions are due to external circumstances that the programmer cannot prevent. For example, an unexpected end of file can be caused by disk error or a broken network connection.
- Example checked exceptions, `IOException`, `SQLException`, etc.

Unchecked exceptions

- The compiler does not require you to keep track of unchecked exceptions.
- Unchecked exceptions are caused by logical errors of the program.
- Example unchecked exceptions, `RunTimeException`, `NullPointerException`, etc.

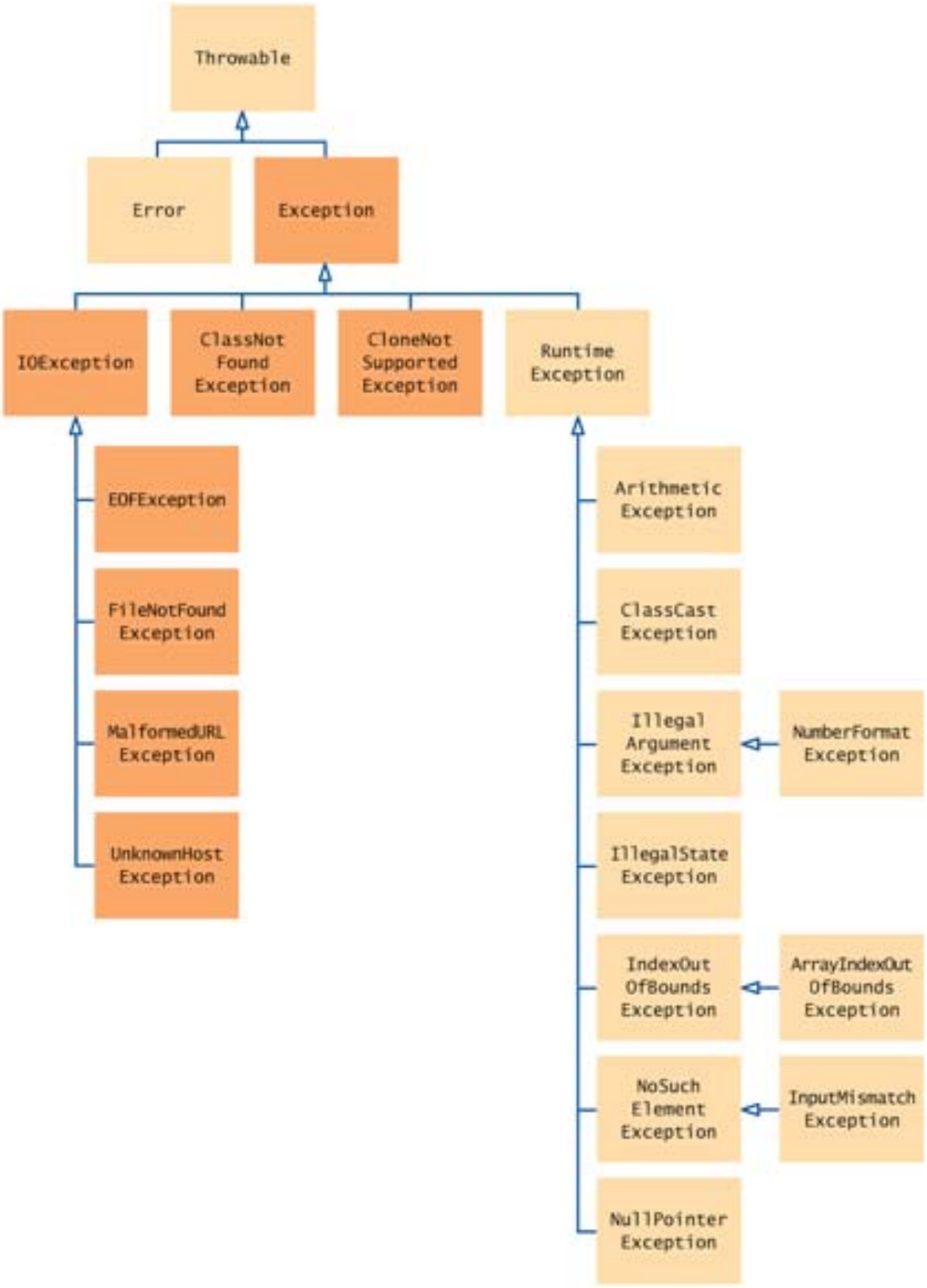


Figure 1 The Hierarchy of Exception Classes

- Categories aren't perfect:
  - `Scanner.nextInt` throws unchecked `InputMismatchException`
  - Programmer cannot prevent users from entering incorrect input
  - This choice makes the class easy to use for beginning programmers
- In general, you need to deal with checked exceptions when programming with files and streams
- For example, use a `Scanner` to read a file:

```
String fname = ...;
FileReader reader = new FileReader(fname);
Scanner in = new Scanner(reader);
```

- But, `FileReader` constructor can throw a `FileNotFoundException` which is a checked exception.

Two ways to handle an exception:

1. Handle the exception, i.e. use the `try/catch` statement.
2. Tell the compiler that you want the method to be terminated (passing control to some other parts of the program) when the exception occurs

Example:

- The program prompts the user to enter a data file name.
- You don't want the program to be terminated if the user mistypes the filename.

```
boolean success = false;
int attempt = 3;  // up to 3 attempts

while (!success && attempt > 0)
{
   attempt--;
   String f = JOptionPane.showInputDialog("Enter filename");

   if (f != null && f.trim().length() > 0)
   {
      try
      {
         Scanner sc = new Scanner(new File(f));
         success = true;
      }
      catch(FileNotFoundException e)
      {
          System.out.println("File not found, try again.");
      }
   }
}

if (success)
   // Statements to make use of the Scanner sc
else
   // do something else
```

Another example:

Consider the `BankAccount` example, a customer is not allowed to withdraw money that exceeds the account balance.

You don't want to abort the banking system because a customer/teller makes an error on the withdrawal slip.

We modify the `withdraw` method as follows:

```
public void withdraw(double amount) throws
                                IllegalArgumentException
{
   if (amount > balance)
       throw new
       IllegalArgumentException("Amount exceeds balance");

   //remaining parts of the method are not executed if
   //an exception has been thrown
   balance = balance – amount;
}
```

The method call that may throw an exception is put in a `try` block

```
try
{
   myAccount.withdraw(10000);
   //some other statements
}
catch (IllegalArgumentException e)
{  // exception handler
   System.out.println("transaction error: " + e);
}
```

- When you throw an exception, the method exits immediately, just as with a return statement.
- Execution does not continue with the method's caller but with an *exception handler* in the catch block.

Example with multiple exceptions:

```
try
{
   String fname = ...; // file name
   FileReader reader = new FileReader(fname); // IOException
   Scanner in = new Scanner(reader);
   String input = in.next();
   int v = Integer.parseInt(input); // NumberFormatException
   ...
}
catch (IOException e)
{
   exception.printStackTrace();
}
catch (NumberFormatException e)
{
   System.out.println("Input was not a number");
}
```

- 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 e)` 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 the exception object `e`

The `finally` clause

- Exception terminates current method
- Danger: Can skip over essential code
- Example:

```
FileReader reader = new FileReader(fname);
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"

```
//the variable reader must be declared outside the try
//block, otherwise it cannot be visible in the finally
//clause
FileReader reader;

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


// Remark:
// new File(fname) does not throw checked exception
// new FileReader(fname) may throw FileNotFoundException
```

```
try
{
    statements;
}
catch (exceptionType1 identifier1)
{
    statements;
}
catch (exceptionType2 identifier2)
{
    statements;
}
...
}
finally
{
    statements;
}
```

- **must include** either one catch clause or a finally clause
- can be multiple catch clauses but **only one** finally clause
- the **try** statements are executed until an exception is thrown or it completes successfully
- a compile-error occurs if the code included in the **try** statement will never throw one of the caught *checked* exceptions
- if an exception is thrown, each **catch** clause is inspected in turn for a type to which the exception can be assigned; be sure to order them from most specific to least specific
- when a match is found, the exception object is assigned to the identifier and the catch statements are executed
- if no matching catch clause is found, the exception percolates up to any outer try block that may handle it
- a catch clause may throw another exception
- if a **finally** clause is included, it's statements are executed after all other try-catch processing is complete
- the finally clause executes whether or not an exception is thrown or a break or continue are encountered

Try-with-resource statement

A resource is an object that must be closed after the program is finished with it.

For example, a file or buffered reader is a resource that should be closed after use.

Prior to Java SE 7, programmer uses a finally block to ensure that a resource is closed.

Starting from Java SE 8, you can use the try-with-resource statement. The declaration statement appears within parentheses immediately after the `try` keyword.

Example codes:

```
static String read_A_Line(String fname)
{
   try (Scanner in = new Scanner(new File(fname)))
   {
      return in.nextLine();
   }
   catch (IOException e)
   { // exception handler }

   // Scanner instance is declared in a try-with-resource
   // statement, it will be closed regardless of whether
   // the try statement completes normally or abruptly.

}
```