As human beings , we commit many errors. A software engineer may also commit several errors while designing the project or developing the code. These errors are also called 'bugs' and the process of removing them is called 'debugging'. Let us take a look at different types of errors that are possible in a program.

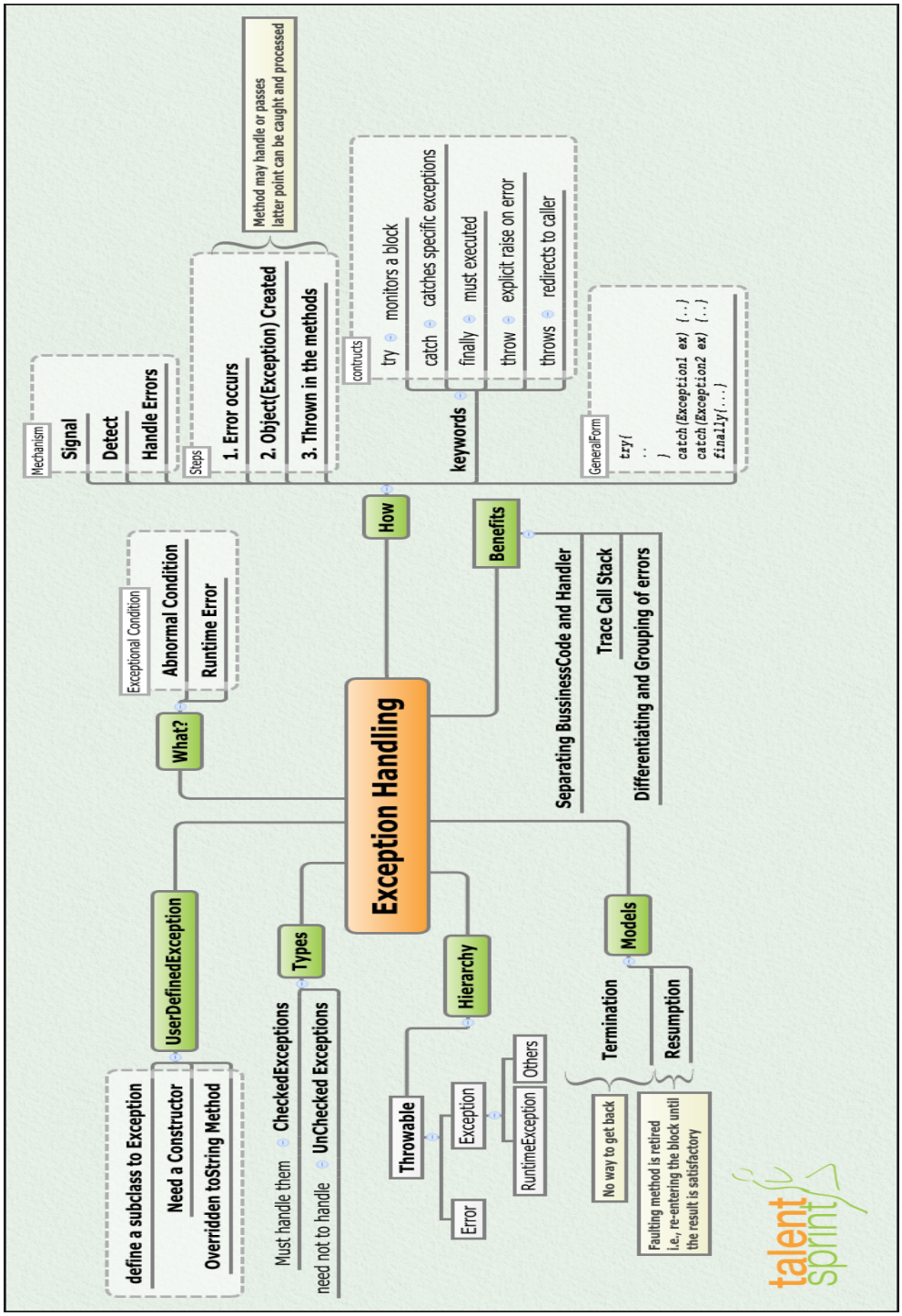
Errors in a Java program

There are basically three types of errors in the java program

Compile-time errors are syntactical errors found in the code, due to which a program fails to compile. For example, forgetting a semicolon at the end of a Java statement, or writing a statement without proper syntax will result in compile-time error.

Run-time errors represent inefficiency of the computer system to execute a particular statement. For example, insufficient memory to store something or inability of the microprocessor to execute some statement come under run-time errors.

Logical errors depict flaws in the logic of the program. The programmer might be using wrong formula or the design of the program itself is wrong. Logical errors are not detected either by Java compiler or JVM. The programmer is soley responsible for them.

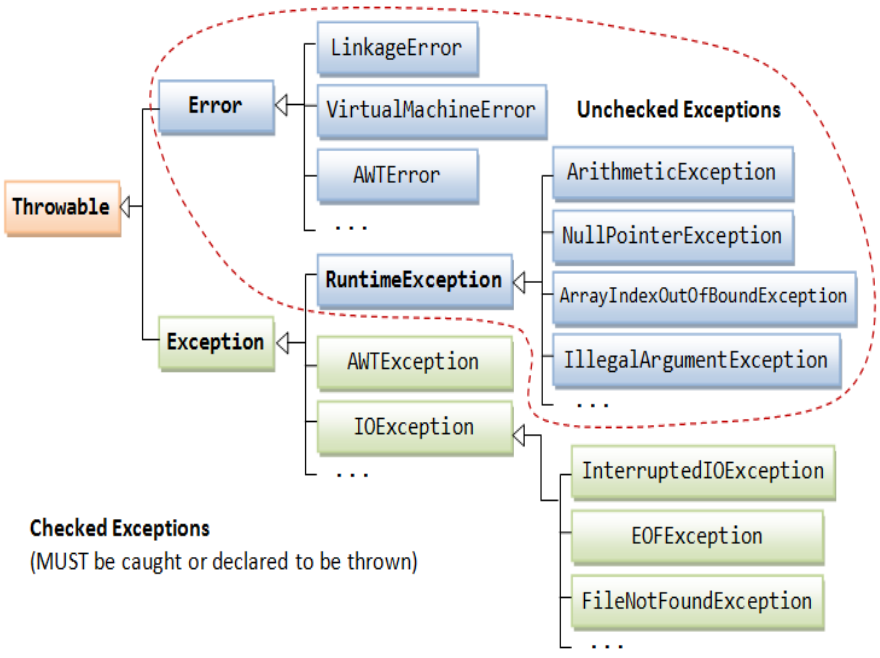


Exceptions

An `Exception` is a runtime error. Then there raises a doubt, can't I call a compile time error an Exception? The answer is , "No you cannot call a compile-time errors as exceptions". They come under errors. All exceptions occur only at runtime but some exceptions are detected at compile time and some others at runtime. The exceptions that are checked at compilation time by the Java compiler are called 'checked exceptions' while the exceptions that are checked by the JVM are called 'unchecked exceptions'.

Unchecked exceptions and errors are considered as unrecoverable and the programmer cannot do any thing when they occur. The programmer can write a Java program with unchecked exceptions and errors and can compile the program. He can see their effect only when he runs the program. So, Java compiler allows him to write a Java program without handling the unchecked exceptions and errors. In case of checked exceptions, the programmer should either handle them or throw them without handling them. He cannot simply ignore them, as Java compiler will remind him of them. Let us now consider a statement

All exceptions are declared as classes in Java. Of course, everything is a class in Java. Even the errors are also represented by classes. All these classes are descended from a super class called **Throwable**, as shown below.



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

Here, IOException is an example for checked exception, it can be thrown out of main() method without handling it. This is done by throws clause written after main() method in the above statement. Of course, we can also handle it. Suppose, we are not handling it and not even throwing it, then the Java compiler will give an error.

The point is that an exception occurs at run time but in case of checked exceptions, whether you are handling it or not, it is detected at compilation time. So let us define an exception as a runtime error that can be handled by a programmer. This means a programmer can do something to avoid any harm caused by the rise of an exception. In case of an error, the programmer cannot do any thing and hence if error happens, it causes some damage.

All exceptions are declared as classes in Java. Even the errors are also represented by classes. All these classes are descended from a super class called `Throwable`.

Generally, in any program, any files or databases are opened in the begining of the program. The data from the file is retrieved and processed in the middle of the program. At the end of the program, the files are closed properly, so that the data in the files is not corrupted. The following program shows the situation.

Example Program that prints `'Welcome'` in the begining. Then the number of command line arguments is assign ed to num. This num divides a number 45 and the result is stored into result. Finally it prints `'Bye'`

```
1  class WHException {
2      public static void main(String args []) {
3          System.out.println (''Welcome'');
4          int num = args.length;
5          System.out.println (''No of args are :'' + num);
6          int result = 45 / num;
7          System.out.println (''The result is :'' + result);
8          System.out.println (''Bye'');
9      }
10 }
```

In the above program, when we pass command line arguments, then the program will execute without any problem. But when we run the program without passing any arguments, then a value become zero. Hence execution of the following expression fails.

int result = 45 / num;

Here, division by zero happens and this value represents infinity. In this case, JVM displays exception details and then terminates the program abnormally. The subsequent statements in the program are not executed. This means any opened files or databases which are open in the program will not be closed and hence the data in the files will be lost. This is the major problem with exceptions. When there is an exception files may not be closed or the threads may abnormally terminate or the memory may not be freed properly. These things lead to many other problems in the software. Closing the opened files, stopping any running threads in the program, and releasing the used memory are called `'cleanup operations'`. Therefore, it is compulsory to design the program in such a way that even if there is an exception, all the clean up operations are performed and only then the program should be terminated. This is called `Exception Handling`.

Exception Handling

When there is an exception, the user data may be corrupted. This should be tackled by the programmer by designing the program carefully. For this, a programmer should perform the following three steps.

1. The programmer should observe the statements in the program where there may be a possibility of exceptions. Such statements should be written inside a `try` block.

   ```
   try {
       statement(s);
   }
   ```

   If some exception arises inside it, the program will not be terminated. When the JVM understand that there is an exception, it stores the exception details in an exception stack and then jumps into `catch` block.

2. The programmer should write the `catch` block where he should display the exception details to the user. This helps the user to understand that there is some error in the program. The programmer should also display a message regarding what can be done to avoid this error.

   ```
   catch (Exception ref) {
       statement(s);
   }
   ```


The reference `ref` is automatically adjusted to refer to the exception stack where the details of the exception stack are available. So, we can display the exception details using any of the following ways

- Using `print()` or `println()` method's, such as System.out.println(ref);
- Using `printStackTrace()` method of Throwable class, which fetches exception details from the exception stack and displays them.

3. The programmer should perform cleanup operations like closing the files and terminating the threads. The programmer should write this code in the finally block.

```
finally {
    statement(s);
}
```

The statements in the finally block are executed irrespective of whether there is an exception or not.

Performing above three tasks is called `'Exception Handling'`. Remember in exception handling the programmer is not preventing the exception, as in many cases it is not possible. But the programmer is avoiding damage that may happen to user data.

Example Program which tells the use of try, catch and finally block.

```
1  class HException {
2      public static void main(String args []) {
3          System.out. println (''Welcome'');
4          try {
5              int num = args.length;
6              System.out. println (''No of args are :''+ num);
7              int result = 45 / num;
8              System.out. println (''The result is :'' + result);
9          }
10         catch(ArithmeticException ae)
11             System.out. println (''cannot divide a number by zero'');
12         }
13          finally {
14             System.out. println (''Bye'');
15         }
```



```
16        }
17  }
```

throws Clause

Even if the programmer is not handling runtime exception, the Java compiler will not give any error related to runtime exceptions. But the rule is that the programmer should handle checked exceptions. In case the programmer does not want to handle the checked exceptions, he should throw them out using `throws` clause. Otherwise, there will be an error flagged by Java compiler.

The following program makes you to unerstand this better, where there is an `IOException` raised by `readLine()` method of `BufferedReader` class. This is checked exception and hence the compiler checks it at the compilation time. It is not handled, the compiler expects at least to throw it out.

Example Program that shows the compile time error for exceptions.

```
1  import java.io.*;
2  class Throws {
3      static void accept() {
4          BufferedReader br = new BufferedReader
5              (new InputStreamReader(System.in));
6          System.out.println ("Enter your Name");
7          String name = br.readLine();
8          System.out.println ("Hello''+''  ''+name);
9      }
10     public static void main(String args[]) {
11         accept();
12     }
13 }
```

In this program, the Java compiler expects the programmer to handle the `IOException` using `try` and `catch` blocks, else `IOException` should be thrown out without handling it. But the programmer is not performing either. So there is a compile time error dispalyed.

Here, we are not handling the `IOException` given by `readLine()` method. Since it is a checked exception, we should throw it out of the method using `throws` clause as

throws IOException

The `throws` clause is written at the side of `accept()` method since in this method the `readLine()` is called. Also, we should write `throws` clause next to `main()` method since in this method, the `accept()` is called as shown here.

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

Now, if the above code is inserted into the program, the preceding program executes without any problem.

Example Program which shows the use of throws clause.

```
1  import java.io.*;
2  class ThrowsEx {
3      static void accept() throws IOException {
4          BufferedReader br = new BufferedReader
5              (new InputStreamReader(System.in));
6          System.out.println ("Enter your Name");
7          String name = br.readLine();
8          System.out.println ("Hello"+"  "+name);
9      }
10     public static void main(String args []) throws IOException {
11         accept();
12     }
13 }
```

throw Clause

There is also a `throw` statement available in Java to throw an exception(predefined or user-defined) explicitly and catch it.

The following program demonstrates the use of **throw** keyword with user-defined exception.

Example

```
1 class LessBalance extends Exception {
2     private String message;
3     public LessBalance(String message) {
4         this.message = message;
```



```
5       }
6       public String toString() {    //it executes when object is used in output
        statement Statement
7           return message;
8       }
9   }
10  class Bank {
11      private int cust_id;
12      private String cust_name;
13      private double balance;
14      public void getDetails(int cust_id, String cust_name, double balance) {
15          try {
16              if (balance < 1000) {
17                  throw new LessBalance("Insufficient Balance");
18              } else {
19                  this.cust_id = cust_id;
20                  this.cust_name = cust_name;
21                  this.balance = balance;
22              }
23          } catch(LessBalance e) {
24              System.out.println("Exception: " + e);
25          }
26      }
27      public void showDetails() {
28          System.out.println("Customer Id: " + cust_id);
29          System.out.println("Customer Name: " + cust_name);
30          System.out.println("Customer Balance : " + balance);
31      }
32  }
33  public class MyException {
34      public static void main(String args[]) {
35          Bank customer1 = new Bank();
36          System.out.println("Details of customer1");
37          customer1.getDetails(100, "Miller", 9999.99);
38          customer1.showDetails();
39          Bank customer2 = new Bank();
```



```
40          System.out.println (''Details of customer2'');
41          customer2.getDetails(101, ''Smith'', 999.99);
42          customer2.showDetails();
43      }
44  }
```

Output

```
$ javac MyException.java
$ java MyException
Details of customer1
Customer Id: 100
Customer Name: Miller
Customer Balance : 9999.99
Details of customer2
Exception: Insufficient Balance
Customer Id: 0
Customer Name: null
Customer Balance : 0.0
$
```

try-with-resources statement

Java SE 7 provides a new feature `try-with-resource` statement that will auto close resources.

```
System.out.println (''About to open'');
try(InputStream in =
 new FileInputStream(''missingfile.txt'')) {
    System.out.println (''File open'');
    int data = in.read();
} catch(FileNotFoundExceptions e) {
    System.out.println (e.getMessage());
} catch(IOException e) {
    System.out.println (e.getMessage());
}
```

The `try-with-resources` statement can eliminate the need for a lengthy finally block. Resources opened using `try-with-resources` statement are always closed. Any class that implements the `java.lang.AutoCloseable` can be used as a resource. If a resource should be autocloseable, its reference must be declared within the `try` statement's parantheses.

Multiple resources can be opened if they are seperated by semocolons. If you open multiple resources, they will be closed in the opposite order in which they are opened.

Catching Multiple Exceptions

A single catch block can handle more than one type of exception. This feature can reduce code duplication.

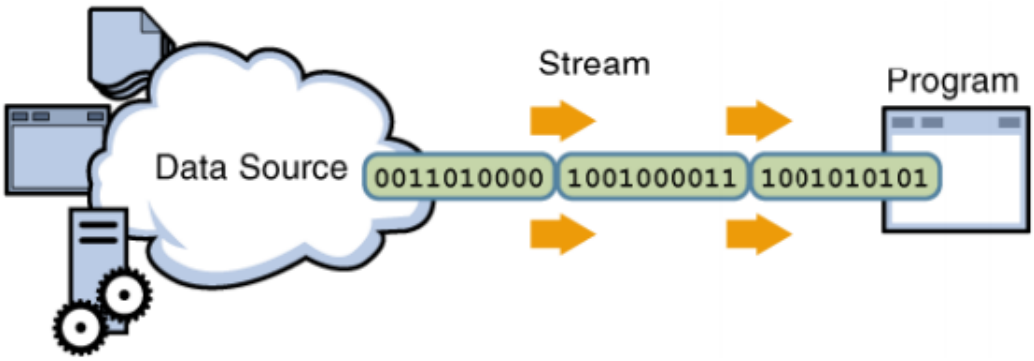
```
catch (IOException | SQLException e) {
    System.out. println (e);
}
```

I/O Streams

- An I/O Stream represents an input source or an output destination.
- A stream can represent many different kinds of sources and destinations like disk files, devices, other programs, a network socket, and memory arrays.
- Streams support many different kinds of data like simple bytes, primitive data types, localized characters, and objects.
- Some streams simply pass on data, others manipulate and transform the data in useful ways.
- No matter how they work internally, all streams present the same simple model to programs that use them. A stream is a sequence of data.

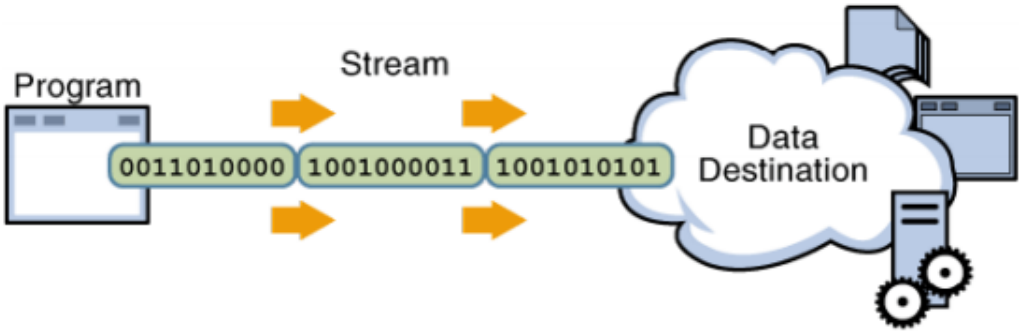
Input Stream

A program uses an input stream to read data from a source, one item at a time.



Output Stream

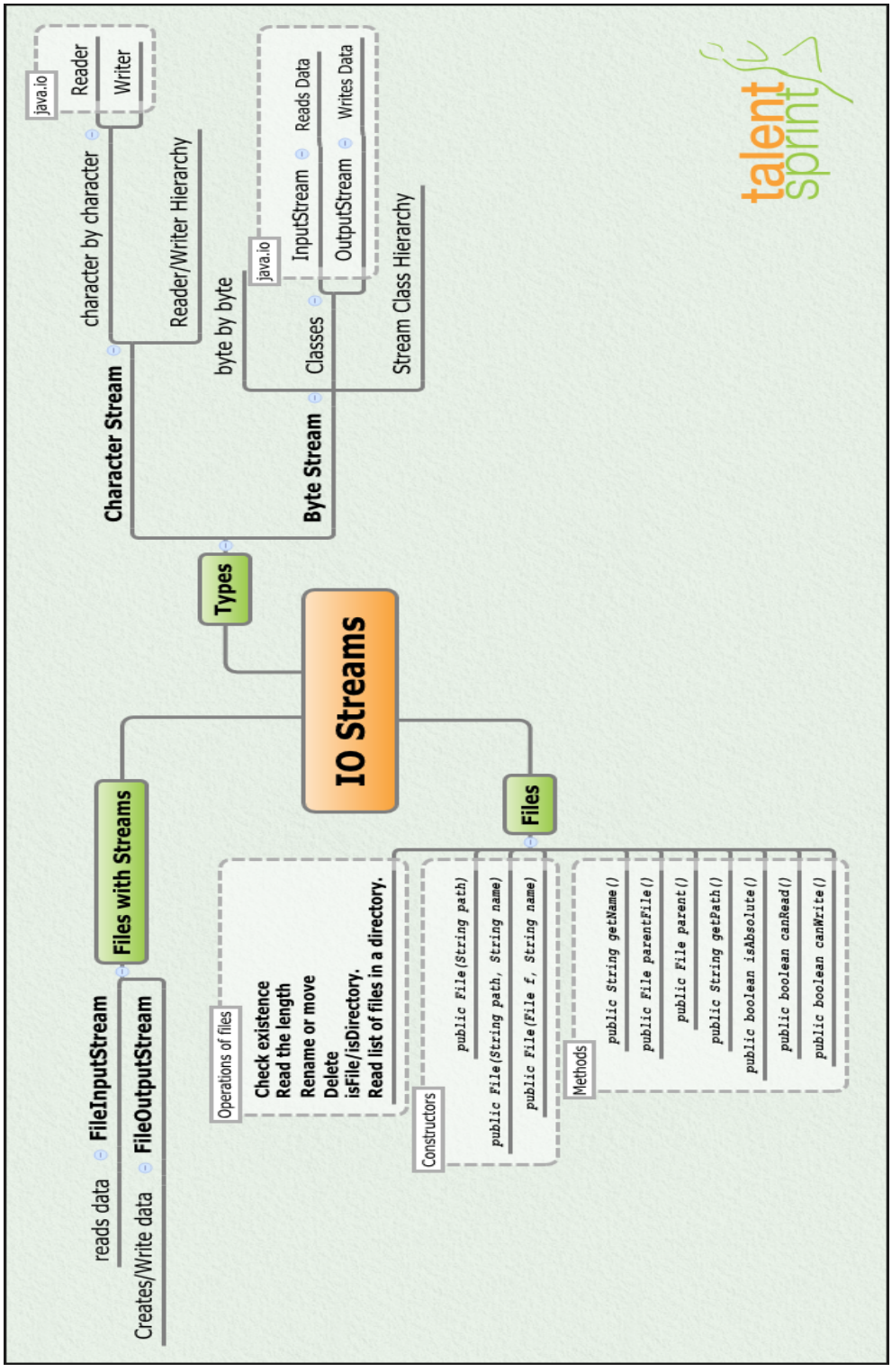
A program uses an output stream to write data to a destination, one item at time.



General Stream Types

The general stream types are:

1. **Character and Byte Streams** Character streams are the streams that read and write 16-bit characters whereas Byte streams are the streams that read and write 8-bit bytes.
2. **Input and Output Streams** Based on source or destination
3. **Node and Filter Streams** Whether the data on a stream is manipulated or transformed or not.



Character and Byte Streams

1. **Byte streams** For binary data, root classes for byte streams are :
   - The InputStream class
   - The OutputStream class
2. **Character streams** For Unicode characters, root classes for character streams are :
   - The Reader class
   - The Writer class

Input and Output Streams

1. **Input or source streams** can read from these streams, root classes of all input streams are :
   - The InputStream class
   - The Reader class
2. **Output or sink (destination) streams** Can write to these streams, root classes of all output streams are :
   - The OutputStream class
   - The Writer class

Node and Filter Streams

1. **Node streams (Data sink stream)** Contain the basic functionality of reading or writing from a specific location, types of node streams include files, memory, and pipes
2. **Filter streams (Processing stream)** Layered onto node streams between threads or processes, used for additional functionality like altering or managing data in the stream
3. Adding layers to a node stream is called `stream chaining`

Control Flow of an I/O Operation

The control flow of an I/O operation is:

- Create a stream object and associate it with a data-source (data-destination)

- Give the stream object with the desired functionality through stream chaining

```
while (there is more information)
    read(write) next data from(to) the stream
close the stream
```

Byte Stream

- Programs use byte streams to perform input and output of 8-bit bytes.
- All byte stream classes are descended from `InputStream` and `OutputStream`.
- There are many byte stream classes like `FileInputStream` and `FileOutputStream`.
- They are implemented in the same way, but they differ mainly in the way they are constructed.

When not to use Byte Stream

Byte Stream represents a kind of low-level I/O that you should avoid:

- If the data contains character data, then the best approach is to use character streams.
- Byte streams should only be used for the most primitive I/O.

Example: FileInputStream and FileOutputStream

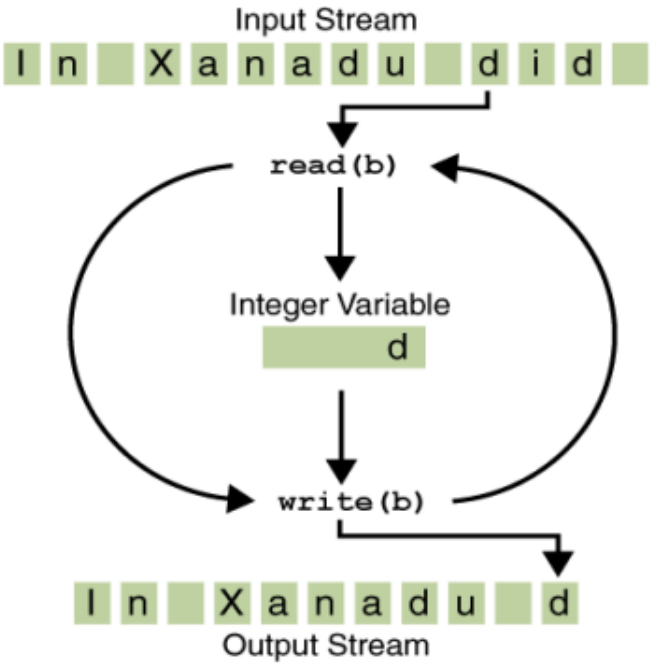
```
1  public class CopyBytes {
2      public static void main(String[] args) throws IOException {
3          FileInputStream in = null;
4          FileOutputStream out = null;
5          try {
6              in = new FileInputStream(''xanadu.txt'');
7              out = new FileOutputStream(''outagain.txt'');
8              int c;
9              while ((c = in.read()) != -1) {
10                 out.write(c);
11             }
12         }
```



```
13            finally {
14                if (in != null) {
15                    in.close();
16                }
17                if (out != null) {
18                    out.close();
19                }
20            }
21        }
22    }
23
```

Simple Byte Stream Input and Output

Simple byte stream input and output is shown in the following diagram:



Character Stream

- The Java platform stores character values using Unicode conventions.

- Character stream I/O automatically translates the internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.
- All character stream classes are descended from `Reader` and `Writer`.
- There are character stream classes of `FileReader` and `FileWriter` that specialize in file I/O.
- Reader and Writer are the abstract superclasses for character streams in `java.io` package.
- The `Reader` and `Writer` classes were added to JDK 1.1 to support internationalization.
- The Reader and Writer classes make it possible to work with character streams rather than byte streams.
- To a large extent, these character-stream classes mirror the byte stream classes, so if you know how to use one, it isn't too difficult to figure out how to use the other.

FileReader and FileWriter Example

```
1  public class CopyCharacters {
2      public static void main(String[] args) throws IOException {
3          FileReader inputStream = null;
4          FileWriter outputStream = null;
5          try {
6              inputStream = new FileReader(''xanadu.txt'');
7              outputStream = new FileWriter(''characteroutput.txt'');
8              int c;
9              while ((c = inputStream.read()) != -1) {
10                 outputStream.write(c);
11             }
12         }
13         finally {
14             if (inputStream != null) {
15                 inputStream.close();
16             }
17             if (outputStream != null) {
```



```
18                  outputStream.close();
19              }
20          }
21      }
22  }
```

Character Stream and Byte Stream

- Character streams are often "wrappers" for byte streams.
- The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. `FileReader`, for example, uses `FileInputStream`, while `FileWriter` uses `FileOutputStream`.

Line-Oriented I/O

- Character I/O usually occurs in bigger units than single characters:
- For the line-oriented I/O, one common single unit is the line that contains a string of characters with a line terminator at the end.
- A line terminator can be a carriage-return or linefeed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n").

Line-oriented I/O Example

```
File inputFile = new File(''farrago.txt'');
File outputFile = new File(''outagain.txt'');
FileReader in = new FileReader(inputFile);
FileWriter out = new FileWriter(outputFile);
BufferedReader inputStream = new BufferedReader(in);
PrintWriter outputStream = new PrintWriter(out);
String l;
while ((l = inputStream.readLine()) != null) {
      System.out.println(l);
       outputStream.println(l);
}
in.close();
out.close();
```


Buffered Streams

- An unbuffered I/O means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, because each such request often triggers disk access, network activity, or some other operation that is relatively expensive.
- To reduce this kind of overhead, the Java platform implements buffered I/O streams:
- Buffered input streams read data from a memory area known as a buffer. The native input API is called only when the buffer is empty
- Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

How to Create Buffered Streams?

- A program can convert an unbuffered stream into a buffered stream using the wrapping idiom. An unbuffered stream object is passed to the constructor for a buffered stream class.

Example:

```
inputStream = new BufferedReader(new FileReader(“xanadu.txt”));
outputStream = new BufferedWriter(new
 FileWriter (“ characteroutput . txt ” ));
```

Buffered Stream Classes

- `BufferedInputStream` and `BufferedOutputStream` create buffered byte streams.
- `BufferedReader` and `BufferedWriter` create buffered character streams.

Flushing Buffered Streams

- It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer
- Some buffered output classes support autoflush, specified by an optional constructor argument:
- When autoflush is enabled, certain key events cause the buffer to be flushed.

- For example, an autoflush `PrintWriter` object flushes the buffer on every invocation of println or format.
- To flush a stream manually, invoke its flush method. The flush method is valid on any output stream, but has no effect unless the stream is buffered.

Standard Streams on Java Platform

- Three standard streams on Java platform are:
  - Standard Input, accessed through System.in
  - Standard Output, accessed through System.out
  - Standard Error, accessed through System.err
- These objects are defined automatically and do not need to be opened.
- `System.out` and `System.err` are defined as `PrintStream` objects

Data Streams

- Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values.
- All data streams implement either the `DataInput` interface or the `DataOutput` interface.
- `DataInputStream` and `DataOutputStream` are the implementations that are applied most widely of these interfaces.

DataOutputStream

`DataOutputStream` can only be created as a wrapper for an existing byte stream object.

```
out = new DataOutputStream(new BufferedOutputStream(
        new FileOutputStream(dataFile)));
for (int i = 0; i < prices.length; i ++) {
    out.writeDouble( prices[i]);
    out. writeInt (units[i]);
    out.writeUTF(descs[i]);
}
```


DataInputStream

- Like DataOutputStream, DataInputStream must be constructed as a wrapper for a byte stream
- End-of-file condition is detected by catching EOFException, instead of testing for an invalid return value.

```
in = new DataInputStream(new BufferedInputStream(
      new FileInputStream(dataFile)));
try {
    double price = in.readDouble();
    int unit = in.readInt();
    String desc = in.readUTF();
} catch (EOFException e) {

}
```

The File Class

- The `File` class is not a stream class.
- The `File` class is important because stream classes manipulate File objects.
- The `File` class is an abstract representation of actual files and directory pathname.

The File Class Example

```
1  import java.io.*;
2  public class FileInfoClass {
3      public static void main(String args[]) {
4          String fileName = args[0];
5          File fn = new File(fileName);
6          System.out.println("Name: " + fn.getName());
7          if (!fn.exists()) {
8              System.out.println(fileName +
9                      " does not exists.");
10
```



```
11              /* Create a temporary directory instead. */
12
13              System.out.println("Creating temp directory");
14              fileName = ``temp'';
15              fn = new File(fileName);
16              fn.mkdir();
17              System.out.println(fileName + (fn.exists() `` exists'': ``does not exist'
    '));
18              System.out.println(`` Deleting temp directory'');
19              fn.delete();
20              System.out.println(fileName + `` is a'' + fn.isFile() `` file.'' :``
    directory.''));
21              if (fn.isDirectory()) {
22                  String content[] = fn.list();
23                  System.out.println(`` The content of
24                          this directory:'');
25                  for (int i = 0; i < content.length; i++) {
26                      System.out.println(content[i]);
27                  }
28              }
29              if (!fn.canRead()) {
30                  System.out.println(fileName + `` is not readable.'');
31                  return;
32              }
33              System.out.println(fileName + `` is ''+fn.length() + `` bytes long.'');
34              System.out.println(fileName + `` is ''+fn.lastModified() + `` bytes
    long.'');
35              if (!fn.canWrite()) {
36                  System.out.println(fileName + `` is not writable.'');
37              }
38          }
39      }
40  }
```


Example

Write a program that illustrates the usage of `InputStreamReader` and `BufferedReader` classes that are used to convert the standard input stream (System.in) from a byte stream to a character stream

```
1  import java.io.*;
2  public class InputConversionApp {
3      public static void main(String args[]) throws IOException {
4          InputStreamReader in = new InputStreamReader(System.in);
5          BufferedReader inStream = new BufferedReader(in);
6          System.out.println("Encoding: " + in.getEncoding());
7          String inputLine;
8          do {
9              System.out.print(">");
10             System.out.flush();
11             inputLine = inStream.readLine();
12             System.out.println(inputLine);
13         } while (inputLine.length() != 0);
14     }
15 }
```

How It Works:

- The InputConversionApp program converts the standard input stream (System.in) from a byte stream to a character stream.
- The input characters are echoed to standard output.
- The program also prints out the encoding that is in effect on your system.

Serialization

- Serialization is an ability to read or write an object to a stream. It is the process of "flattening" an object.
- Serialization is used to save object to some permanent storage. Its state should be written in a serialized form to a file such that the object can be reconstructed at a later time from that file.

- Serialization is used to pass on to another object by the OutputStream class and can be sent over the network.

Streams Used for Serialization

The streams used for serialization are:

ObjectOutputStream For serializing (flattening an object)

ObjectInputStream For deserializing (reconstructing an object)

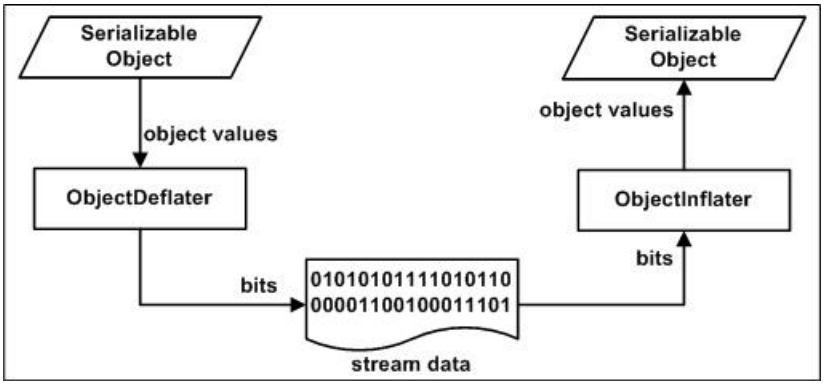


Figure 1: Serialization and Deserialization

Requirement for Serialization

- Serialization is required to allow an object to be serializable
- Its class should implement the `Serializable` interface
  - `Serializable` interface is marker interface
  - Its class should also provide a default constructor (a constructor with no arguments)
- Serializability is inherited
  - Do not have to implement Serializable on every class
  - Can just implement Serializable once along the class hierarchy

Non-Serializable Objects

- Most Java classes are serializable.
- Objects of some system-level classes are not serializable:
- Because the data that they represent, changes constantly:
- Reconstructed object will contain different value anyway
- For example, thread running in your JVM would be applying the memory of your system. Persisting it and trying to run it in your JVM would make no sense at all
- A `NotSerializableException` is thrown if you try to serialize non-serializable objects

What is Preserved when an Object is Serialized?

When an object is serialized enough information is preserved that is needed to reconstruct the object instance at a later time:

- Only the data of the object is preserved
- Methods and constructors are not part of the serialized stream
- Class information is included

The transient Keyword

- The `transient` modifier applies only to instance variables.
- If you mark an instance variable as `transient`, you are telling the JVM to skip (ignore) this variable when you attempt to serialize the object containing it.
- In other words, the `transient` keyword prevents the data from being serialized.
- All fields that are not `transient` are considered part of the persistent state of an object and are eligible for persistence.

Serialization: Writing an Object Stream

To write an object stream use its writeObject method of the ObjectOutputStream class.

public final void writeObject(Object obj) **throws** IOException

where obj is the object to be written to the stream


```
1  import java.io.*;
2  public class Employee implements Serializable {
3      public String name;
4      public String address;
5      public transient int SSN;
6      public int id;
7      public void mailCheck() {
8          System.out.println("Mailing a check to " + name + " " + address);
9      }
10 }
```

```
1  public class SerializeDemo {
2      public static void main(String [] args) {
3          Employee e = new Employee();
4          e.name = "Reyan Ali";
5          e.address = "Phokka Kuan, Ambehta Peer";
6          e.SSN = 11122333;
7          e.number = 101;
8          try {
9              FileOutputStream fileOut = new FileOutputStream("employee.ser");
10             ObjectOutputStream out = new ObjectOutputStream(fileOut);
11             out.writeObject(e);
12             out.close();
13             fileOut.close();
14             System.out.print("Serialized data is saved in employee.ser");
15         } catch(IOException i) {
16             i.printStackTrace();
17         }
18     }
19 }
```

Deserialization: Reading an Object Stream

To read an object stream use its `readObject` method of the `ObjectInputStream` class.

```
public final Object readObject() throws IOException,
                        ClassNotFoundException
```


where 'obj' is the object to be read from the stream

The Object type returned should be typecasted to the appropriate class name before methods on that class can be executed.

```
1  import java.io.*;
2  public class DeserializeDemo {
3      public static void main(String [] args) {
4          Employee e = null;
5          try {
6              FileInputStream fileIn = new FileInputStream("employee.ser");
7              ObjectInputStream in = new ObjectInputStream(fileIn);
8              e = (Employee) in.readObject();
9              in.close();
10             fileIn.close();
11         } catch(IOException i) {
12             i.printStackTrace();
13             return;
14         } catch(ClassNotFoundException c) {
15             System.out.println("Employee class not found");
16             c.printStackTrace();
17             return;
18         }
19         System.out.println("Deserialized Employee...");
20         System.out.println("Name: " + e.name);
21         System.out.println("Address: " + e.address);
22         System.out.println("SSN: " + e.SSN);
23         System.out.println("Number: " + e.number);
24     }
25 }
```

