Java Application Design

Exceptions & IO

Weng Kai

pitfall

```
System.out.println(
true?Integer.valueOf(1):Double.valueOf(2));
```

Run-time Error

- The basic philosophy of Java is that "badly formed code will not be run."
- There's always something happen in run-time.
- It is very important to deal with all possible situation in the future running.

read a file

open the file;

determine its size;

allocate that much memory;

read the file into memory;

close the file;

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if ( theFilesOpen ) {
        determine its size;
        if ( gotTheFileLength ) {
            allocate that much memory;
            if ( gotEnoughMemory ) {
                read the file into memory;
                if ( readFailed ) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if ( theFlLEDidntClose && errorCode == 0 ) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Working w/ exception

```
try {
   open the file;
   determine its size;
   allocate that much memory;
   read the file into memory;
   close the file;
} catch ( fileOpenFailed ) {
   doSomething;
} catch ( sizeDeterminationFailed ) {
   doSomething;
} catch ( memoryAllocationFailed ) {
   doSomething;
} catch ( readFailed ) {
   doSomething;
} catch ( fileCloseFailed ) {
   doSomething;
}
```

exception

- I take exception to that
- At the point where the problem occurs, you might not know what to do with it, but you do know that you can't just continue on merrily; you must stop, and somebody, somewhere, must figure out what to do.

Why exception?

- The significant benefit of exceptions is that they clean up error handling code.
- It separates the code that describes what you want to do from the code that is executed.

Basic exceptions

- An exceptional condition is a problem that prevents the continuation of the method or scope that you're in.

```
if(t == null)
  throw new NullPointerException();
```

- or

```
throw new
  NullPointerException("t = null");
```

throw

1. First use new to create an object that represents the error condition.
2. Give the resulting reference to **throw**.
3. The object is, in effect, "returned" from the method.
4. An exception mechanism raised up to handle the exception object.

What can be thrown?

- Any type of Throwable (the exception root class) object that you want.

Catching an exception

```
try {
  // Code that might generate exceptions
} catch(Type1 id1) {
  // Handle exceptions of Type1
} catch(Type2 id2) {
  // Handle exceptions of Type2
} catch(Type3 id3) {
  // Handle exceptions of Type3
}
```

The catch mechanism

- When an exception raised, all statements followed in the same scope are to be disposed. Then it checks the surrounding scope first:

- Is surrounding a try?
  - NO: leave that scope
    - Is it a method body?
      - YES: return to the caller, then
      - go to 1st line
  - YES: try to match a catch
    - match?
      - YES: do the clause and go to the statement followed by try block
      - NO: go to 1st line

```
void m() {
  try {
    f();
  } catch (B b) {
  }
  System.out....
}
void f() {
  try {
    g();
    int i=0;
  } catch (A a) {
  }
}
void g() {
  if ( a == null ) {
    throw new B();
    System.....
  }
  System.out....
}
```

Match the exception

- Is-A
- Case: Human.java

Create your own exceptions

- Case: SimpleExceptionDemo.java

The exception specification

- You're required to inform the client programmer, who calls your method, of the exceptions that might be thrown from your method.

```
void f() throws TooBig, TooSmall, DivZero { //...

void f() {..
```

- You can claim to throw an exception that you really don't.

The exception specification (con't)

- If your method calls a method claims any exceptions, you must either:
  - put the call in a try and set a catch for all the exceptions it claims, or
  - declare to throw those un-handled exceptions.

Catching any exception

```
catch(Exception e) {
  System.err.println("Caught an exception");
}
```

Interface: throwable

- String getMessage();
- String toString();
- void printStackTrace();
- void printStackTrace(PrintStream);

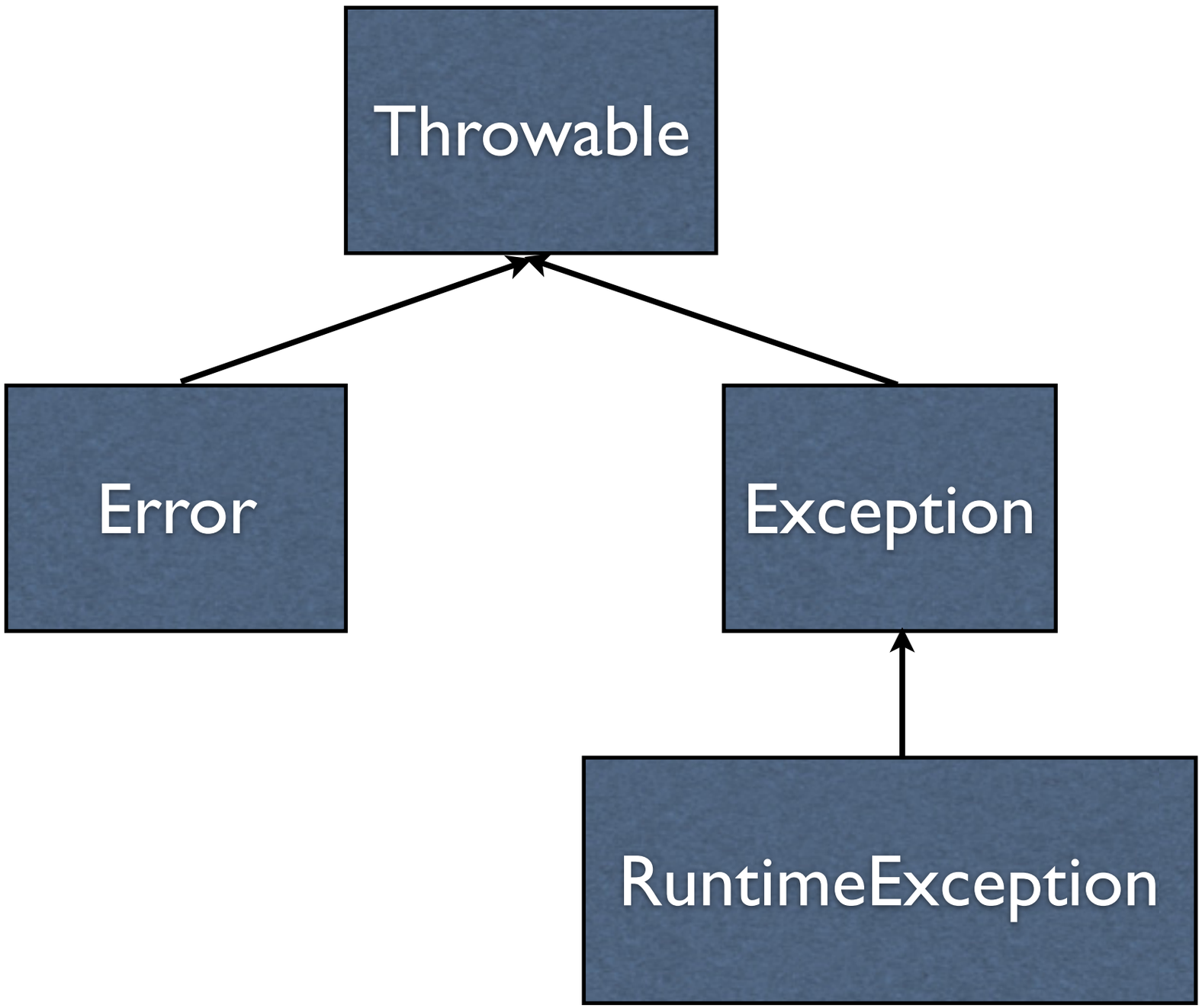
- Case Study : ExceptionMethods.java

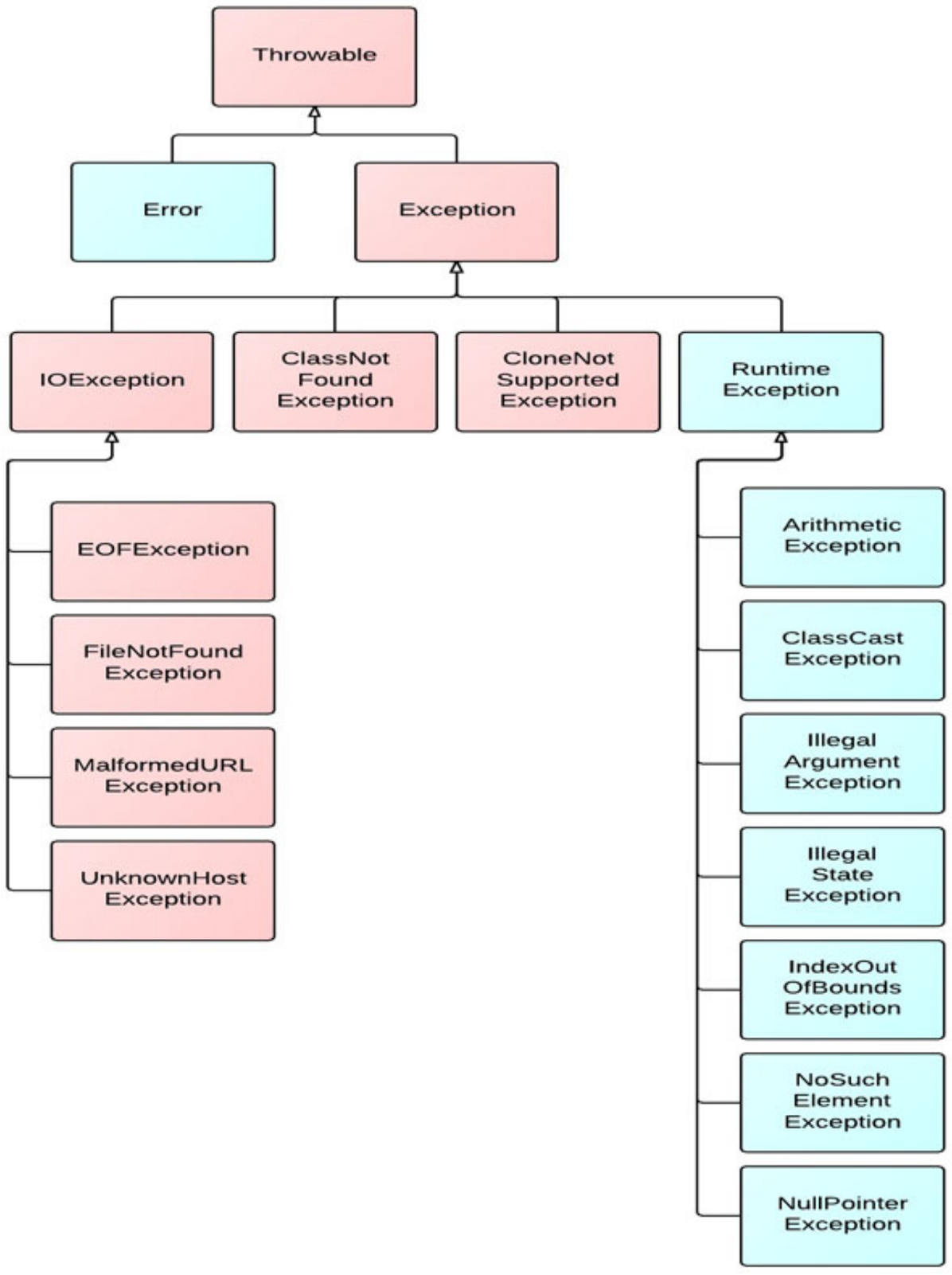
Re-throw an exception

```
catch(Exception e) {
  System.err.println("An exception was thrown");
    throw e;
  }
```

- case: Rethrowing.java

Standard Java exceptions




Throwable
Error
Exception
IOException
ClassNot Found Exception
CloneNot Supported Exception
Runtime Exception
EOFException
FileNotFound Exception
MalformedURL Exception
UnknownHost Exception
Arithmetic Exception
ClassCast Exception
Illegal Argument Exception
Illegal State Exception
IndexOut OfBounds Exception
NoSuch Element Exception
NullPointer Exception

RuntimeException

- You don't have to declare if your method may throw a RuntimeException.
- If you don't provide a try-catch for them...
- Case: NeverCaught.java

Performing cleanup with finally

```
try {
  // The guarded region: Dangerous activities
  // that might throw A, B, or C
} catch(A a1) {
  // Handler for situation A
} catch(B b1) {
  // Handler for situation B
} catch(C c1) {
  // Handler for situation C
} finally {
  // Activities that happen every time
}
```

Three possible leave

- Nothing happened
- An exception thrown, and caught
- An exception thrown, but not caught

- execute finally clause b4 leave
- case: FinallyWorks.java

What's finally for?

- The finally clause is necessary when you need to set something other than memory back to its original state.

Exception restrictions

- When you override a method, you can throw only the exceptions that have been specified in the base-class version of the method.

- case: StormyInning.java

Constructors

- Memory is safe in Java
- Do other clear job yourself with finally clause

try-finally

```
FileInputStream inputStream = null;
  try {
      inputStream = new FileInputStream(new File("test"));
      System.out.println(inputStream.read());
  } catch (IOException e) {
      throw new RuntimeException(e.getMessage(), e);
  } finally {
      if (inputStream != null) {
          try {
              inputStream.close();
          } catch (IOException e) {
              throw new RuntimeException(e.getMessage(), e);
          }
      }
  }
```

- To close an opened resource safely, finally clause may be the choice.

Try-with-resource

```
  try (FileInputStream inputStream = new
FileInputStream(new File("test"))) {
      System.out.println(inputStream.read());
  } catch (IOException e) {
      throw new RuntimeException(e.getMessage(), e);
  }
```

- inputStream will be closed automatically
- The resource can be put here must implements AutoCloseable

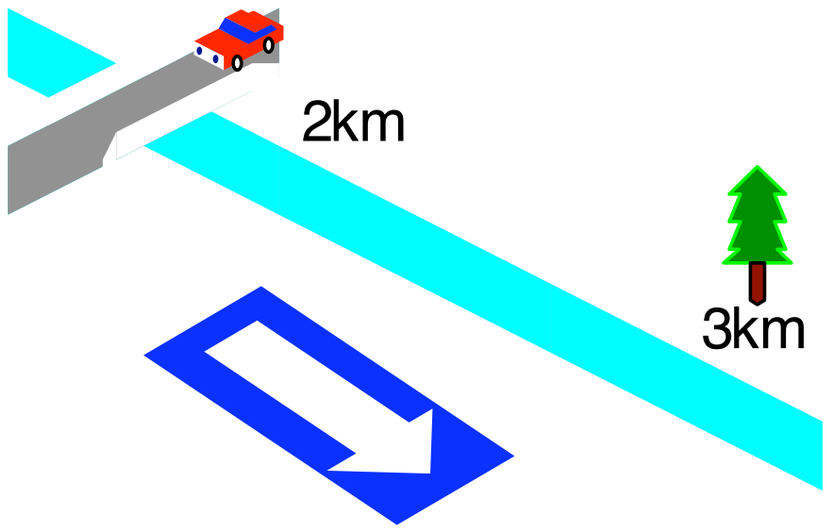
Try-with-resource 9

```
String readData(String message) throws IOException {
    Reader inputString = new StringReader(message);
    BufferedReader br = new BufferedReader(inputString);
    try (br) {
        return br.readLine();
    }
}
```

- The variable here must be final or final de facto

Stream

- Creating a good input/output (I/O) system is one of the more difficult tasks for the language designer.
- The basic idea of Java IO is stream, which is one direction and one dimension.



InputStream

FileInputStream

ByteArrayInputStream

PipedInputStream

FilterInputStream

DataInputStream

LinuNumberInputStream

BufferedInputStream

OutputStream

FileOutputStream

ByteArrayOutputStream

PipedOutputStream

FilterOutputStream

DataOutputStream

PrintStream

BufferedOutputStream

InputStream

- read()
  - int read()
  - read(byte b[])
  - read(byte[],int off,int len)
- skip(long n)
- int available()
- mark()
- reset()
- boolean markSupported()
- close()
- case: GetInput.java

Types of InputStream

- ByteArrayInputStream
- StringBufferInputStream
- FileInputStream
- PipedInputStream
- SequenceInputStream

OutputStream

- write()
  - write(int b)
  - write(byte b[])
  - write(byte b[],int off,int len)
- flush()
- close()

Types of OutputStream

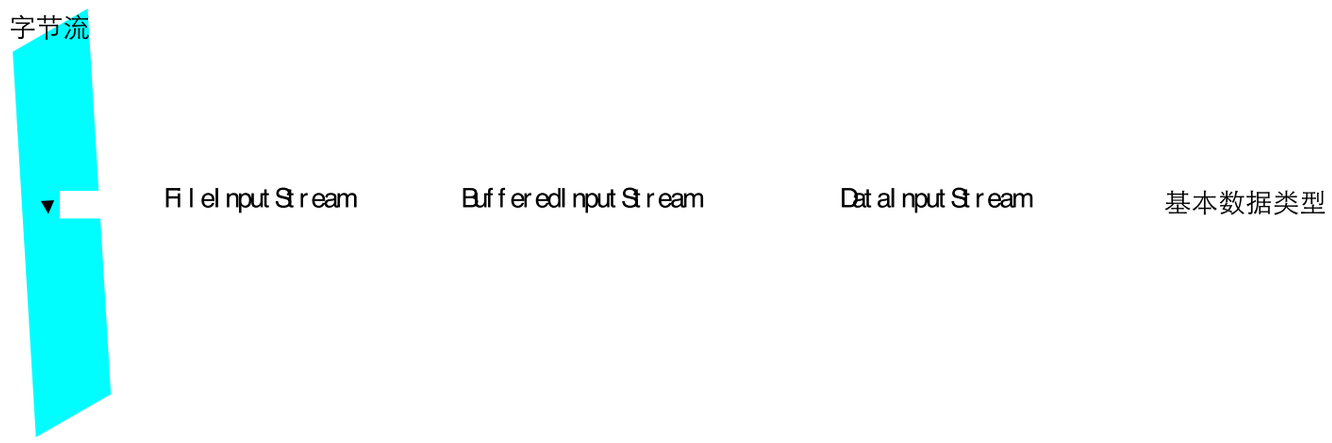
- ByteArrayOutputStream
- FileOutputStream
- PipedOuputStream

Case Study: FileApp.java, FileApp2.java

Filtered Stream

- DataInputStream
- LineNumberInputStream
- BufferedInputStream
- PushbackInputStream

- DataOutputStream
- PrintStream
- BufferedOutputStream



Case: DataSave.java

The new IO:Reader & Writer

- 8-bit vs. 16-bit

Case Study: ReaderWriter.java,
NewIODemo.java

old or new?

- For binary files, use input/output streams.
- For local char. coding text files, use LineNumberReader/PrintWriter based on Input/OutputStream.

```
PrintWriter pw = new PrintWriter(
    new BufferedWriter(
        new OutputStreamWriter(
            new FileOutputStream("abc.txt"))));
```

- For Unicode char. text files, use FileReader/FileWriter directly.

Class File is not file

- It can represent either the name of a particular file or the names of a set of files in a directory.
- Case:

1. DirList.java,
2. DirList2.java,
3. DirList3.java

- Case: MoreAll.java
- Case: MakeDirectories.java

Formatted output

- For PrintStream and PrintWriter
  - public PrintStream format(String format, Object... args)
    - public PrintStream printf(String format, Object... args)

- case: Root.java

Formatted input

- java.util.Scanner
- A simple text scanner which can parse primitive types and strings using regular expressions.
- 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.

```
Scanner sc = new Scanner(System.in);

int i = sc.nextInt();


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

Case: Cash2.java

Compress

- Case Study: GZIPcompress.java

- About JAR

ar [options] destination [manifest] inputfile(s)

jar cf myJarFile.jar *.class

jar tf myJarFile.jar

CLASSPATH="lib1.jar;lib2.jar;"

- allows you to take any object that implements the Serializable interface and turn it into a sequence of bytes that can later be fully restored to regenerate the original object.
- To serialize an object, you create some sort of OutputStream object and then wrap it inside an ObjectOutputStream object. At this point you need only call writeObject( ), and your object is serialized and sent to the OutputStream. To reverse the process, you wrap an InputStream inside an ObjectInputStream and call readObject( ). What comes back is, as usual, a reference to an upcast Object, so you must downcast to set things straight.
- Case: Worm.java