Java I/O, files etc

Mārtiņš Leitass

Agenda

- Command line arguments
- System properties
- I/O stream fundamentals
- InputStreamReader and OutputStreamWriter
- Object serialization/deserialization
- Console I/O
- java.io.File class
- Java 7 enhancements

Agenda

- Command line arguments
- System properties
- I/O stream fundamentals
- InputStreamReader and OutputStreamWriter
- Object serialization/deserialization
- Console I/O
- java.io.File class
- Java 7 enhancements

Command-Line Arguments

- Any Java technology application can use command-line arguments.
- These string arguments are placed on the command line to launch the Java interpreter after the class name:

```
java TestArgs arg1 arg2 «another arg»
```

- Each command-line argument is placed in the args array that is passed to the static main method:

```
public static void main(String[] args)
```

Command-Line Arguments

```
1    public class TestArgs {
2       public static void main(String[] args) {
3          for ( int i = 0; i < args.length; i++ ) {
4             System.out.println("args[" + i + "] is '" + args[i] + "'");
5          }
6       }
7    }
```

Example execution:

java TestArgs arg0 arg1 "another arg"
args[0] is 'arg0'
args[1] is 'arg1'
args[2] is 'another arg'

Command line arguments: args[] content?

- :>java MyClass my command line arguments

0:my 1:command 2:line 3:arguments

- :>java MyClass my “command line” arguments

0:my 1:command line 2:arguments

- :>java MyClass my “command” “line arguments”

0:my 1:command 2:line arguments

- :>java MyClass my “command” “line arguments\”

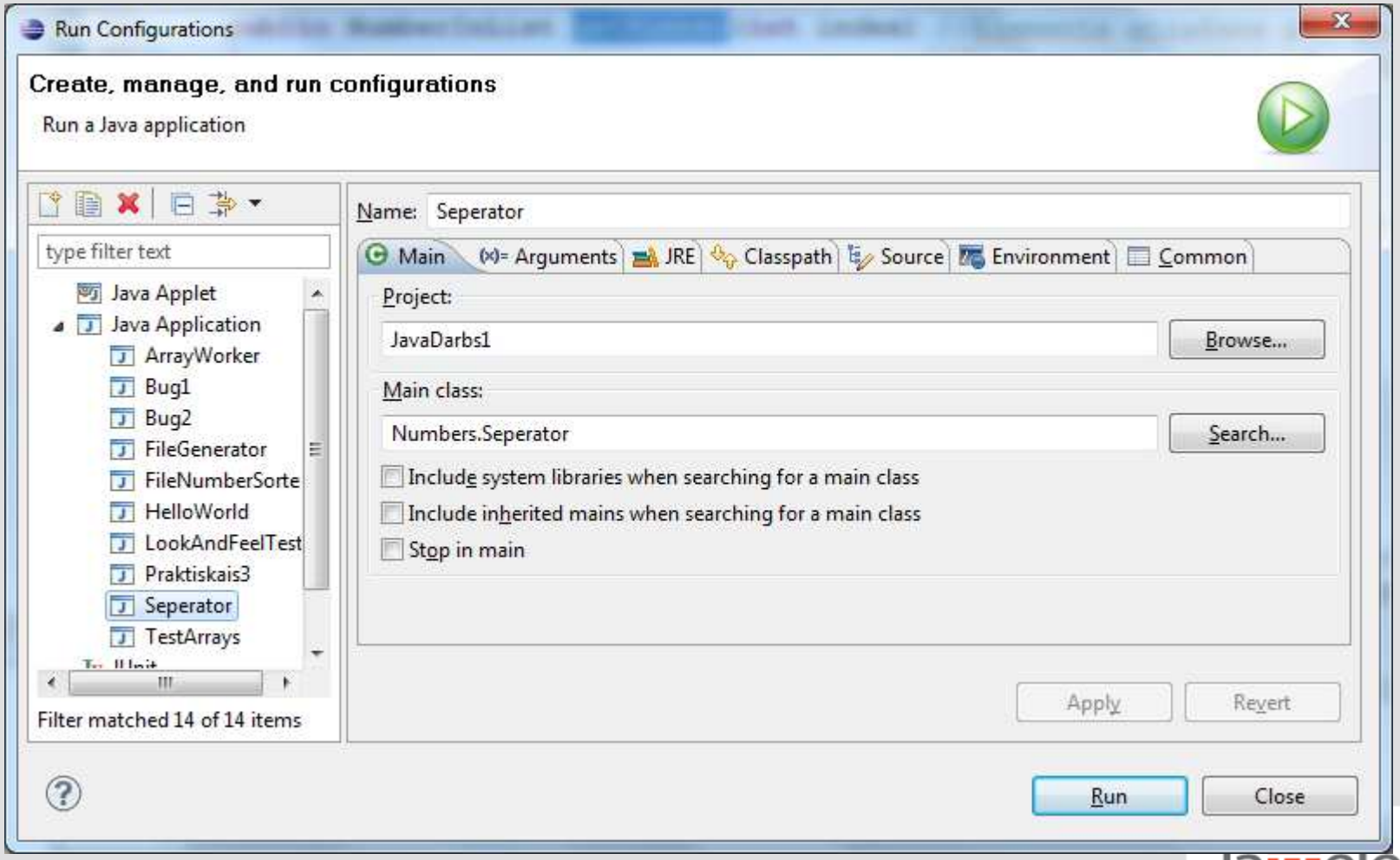
0:my 1:“command” 2:“line 3:arguments\”

- :>java MyClass propertyOne="x" propertyTwo=y

0:propertyOne=x 1:propertyTwo=y

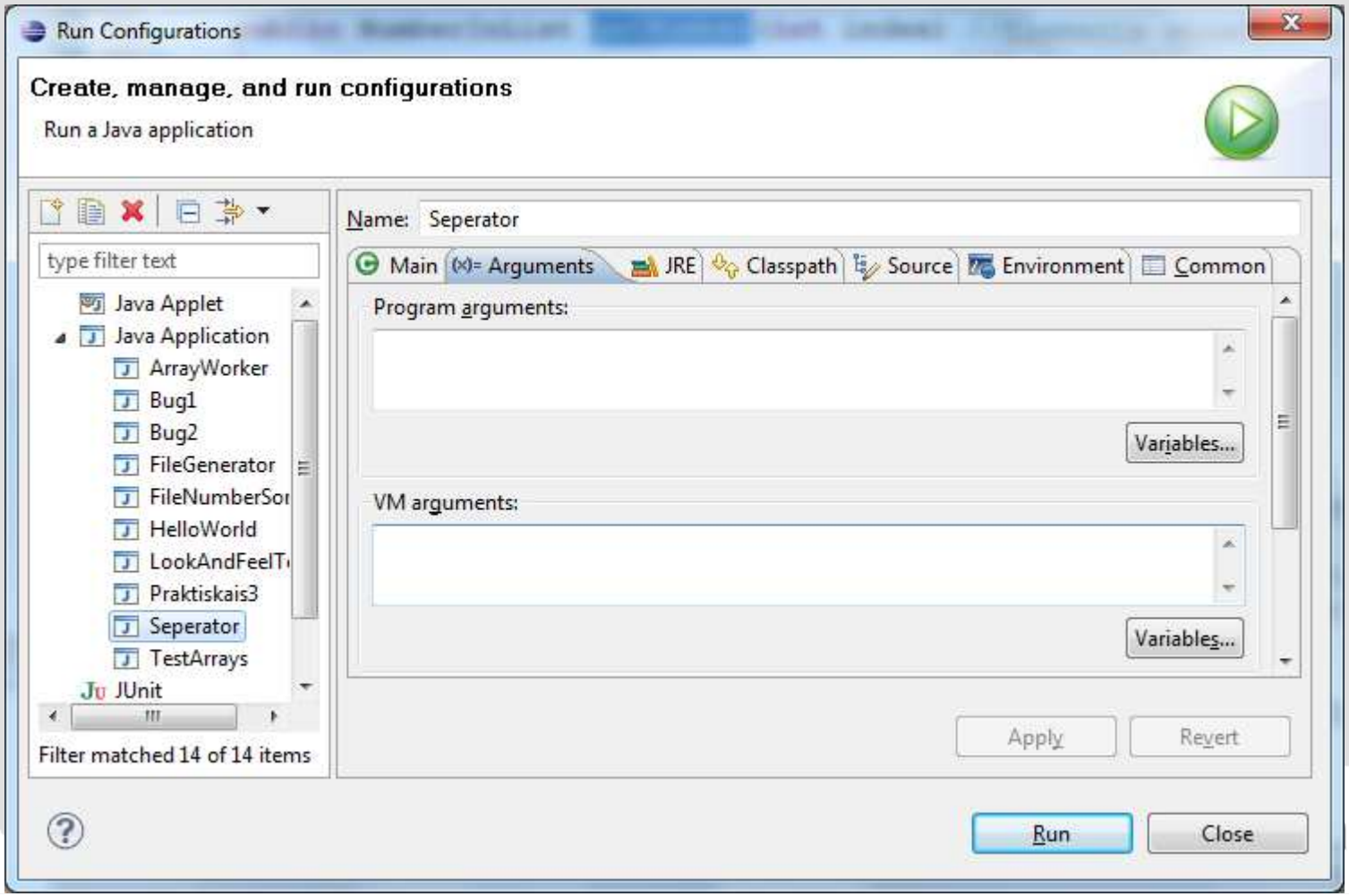
Passing arguments in Eclipse

- Run->Run Configurations...



Passing arguments in Eclipse

- Tab «Main»: set project and main class
- Tab «Arguments»: set program and VM arguments



Agenda

- Command line arguments
- System properties
- I/O stream fundamentals
- InputStreamReader and OutputStreamWriter
- Object serialization/deserialization
- Console I/O
- java.io.File class
- Java 7 enhancements

System properties

- System properties are a feature that replaces the concept of *environment variables* (which are platform-specific).
- The System.getProperties method returns a Properties object.
- The getProperty method returns a String representing the value of the named property.
- Use the -D option on the command line to include a new property.

The Properties Class

- The `Properties` class implements a mapping of names to values (a String-to-String map).
- The `propertyNames` method returns an *Enumeration* of all property names.
- The `getProperty` method returns a String representing the value of the named property.
- You can also read and write a properties collection into a file using load and store.

The Properties Class

```
1    import  java.util.Properties;
2    import  java.util.Enumeration;
3
4    public  class  TestProperties  {
5       public  static  void  main(String[]  args)  {
6          Properties  props  =  System.getProperties();
7          props.list(System.out);
8       }
9    }
```

The Properties Class

- The following is an example test run of this program:

```
java  -DmyProp=theValue  TestProperties
```

- The following is the (partial) output:

```
java.runtime.name=Java(TM)  SE  Runtime  Environment
sun.boot.library.path=C:\jse\jdk1.6.0\jre\bin
java.vm.version=1.6.0-b105
java.vm.vendor=Sun  Microsystems  Inc.
java.vm.name=Java  HotSpot(TM)  Client  VM
file.encoding.pkg=sun.io
user.country=US
myProp=theValue
```

Reading properties from file

```
Properties someProperties = null;
FileInputStream fin = null;
try{
        someProperties = new Properties();
        fin = new FileInputStream(“file.properties");
        someProperties.load(fin);
} catch (Exception ex){
        //apstrādā kļūdu
} finally {
        try{
                fin.close();
        } catch (Exception ex){};
        fin = null;
}
```

Agenda

- Command line arguments
- System properties
- I/O stream fundamentals
- InputStreamReader and OutputStreamWriter
- Object serialization/deserialization
- Console I/O
- java.io.File class
- Java 7 enhancements

Streams

- All modern I/O is stream-based
- A stream is a connection to a source of data or to a destination for data (sometimes both)
- An input stream may be associated with the keyboard
- An input stream or an output stream may be associated with a file
- Different streams have different characteristics:
  - A file has a definite length, and therefore an end
  - Keyboard input has no specific end

The I/O Package

- The java.io package defines I/O in terms of streams.
- The java.nio package and its subpackages define I/O in terms of buffers and channels. Here the “nio” is acronym of new I/O.
- The java.net package provides specific support for network I/O, based around the use of sockets, with an underlying stream or channel-based model.

Fundamental Stream Classes

Stream	Byte Streams	Character Streams
Source streams	InputStream	Reader
Sink streams	OutputStream	Writer

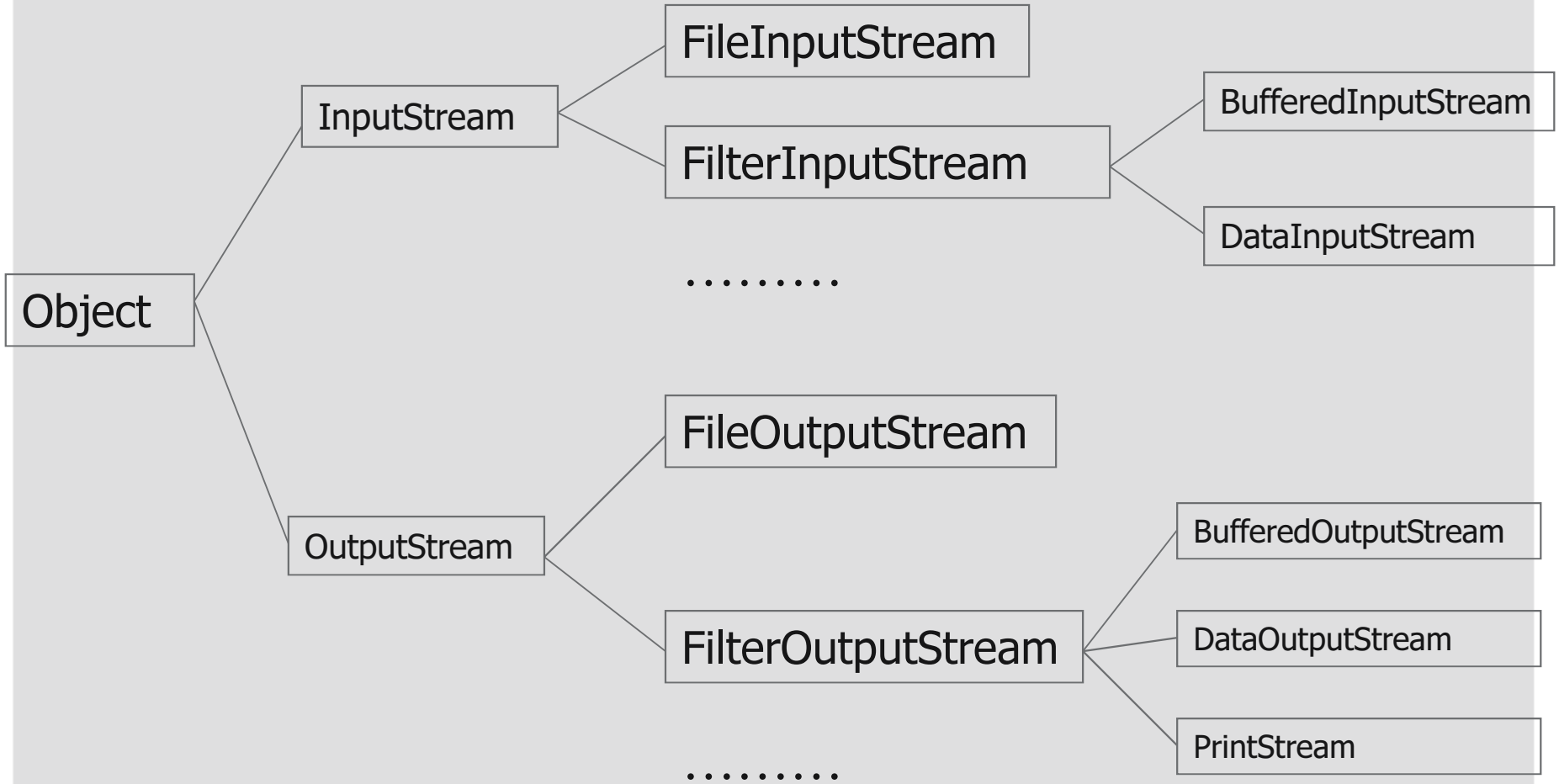
Data Within Streams

- Java technology supports two types of streams: character and byte.
- Input and output of character data is handled by readers and writers.
- Input and output of byte data is handled by input streams and output streams:
  - Normally, the term *stream* refers to a byte stream.
  - The terms *reader* and *writer* refer to character streams.

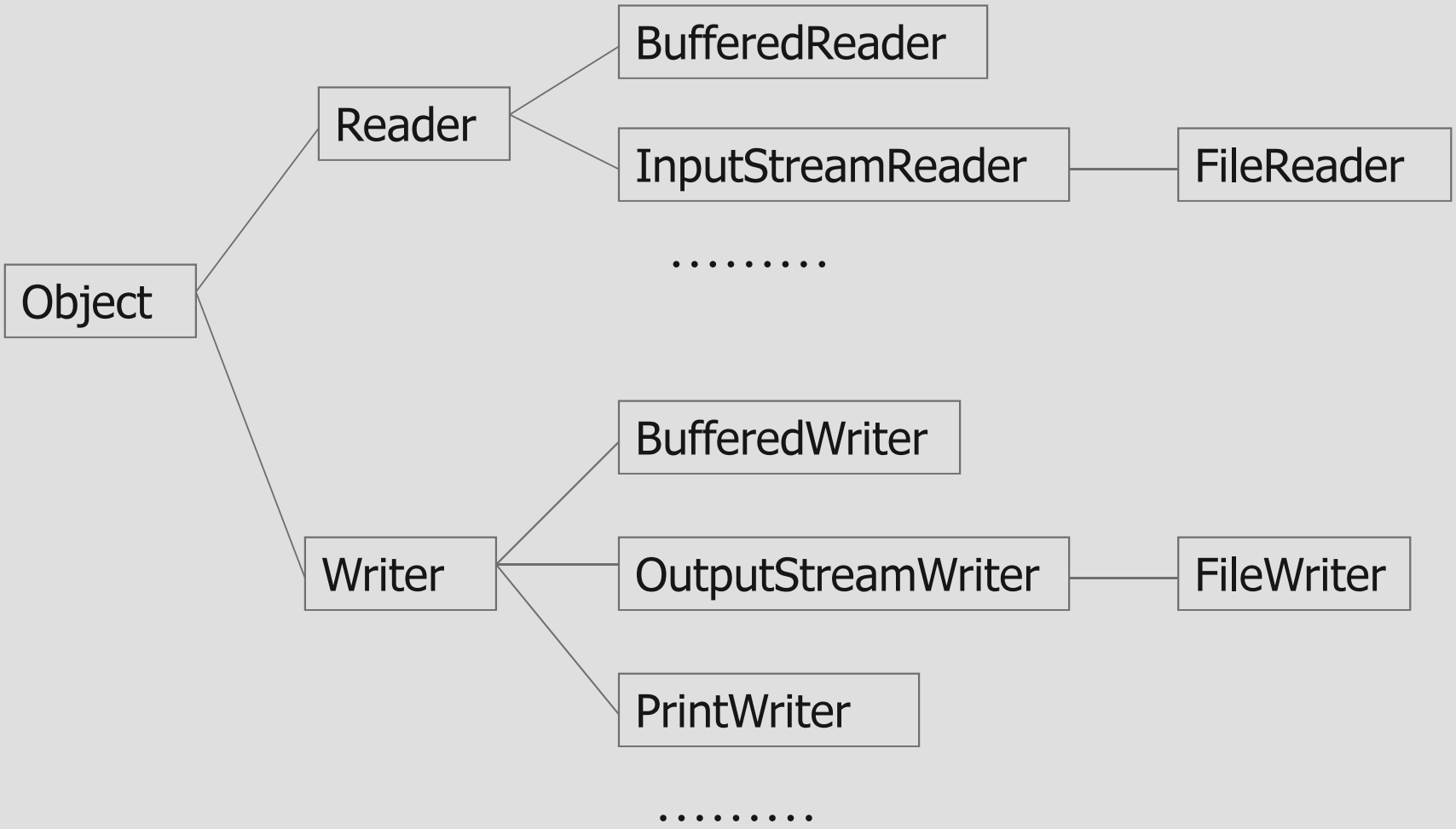
Streams Overview

- Two Major parts in the package java.io : character(16-bit UTF-16 characters) streams and byte(8 bits) streams
- I/O is either text-based or data-based (binary)
- Input streams or output streams → byte stream
- Readers or Writers → character streams
- Five group of classes and interfaces in java.io
  - The general classes for building different types of byte and character streams.
  - A range of classes that define various types of streams – filtered, piped, and some specific instances of streams
  - The data stream classes and interfaces for reading and writing primitive values and strings.
  - For Interacting with files
  - For the object serialization mechanism

Byte (Binary) Streams



Character Streams



Opening a stream

- There is data external to your program that you want to get, or you want to put data somewhere outside your program
- When you open a stream, you are making a connection to that external place
- Once the connection is made, you forget about the external place and just use the stream

The InputStream Methods

- The three basic read methods are:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close()
int available()
long skip(long n)
boolean markSupported()
void mark(int readlimit)
void reset()
```

The OutputStream Methods

- The three basic write methods are:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close()
void flush()
```

The Reader Methods

- The three basic read methods are:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

- Other methods include:

```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

The Writer Methods

- The basic write methods are:

```
void   write(int  c)
void   write(char[]      cbuf)
void   write(char[]      cbuf, int offset, int length)
void   write(String      string)
void   write(String      string, int offset, int length)
```

- Other methods include:

```
void  close()
void  flush()
```

A Simple Example

This program performs a copy file operation using a manual buffer:

java TestNodeStreams file1 file2

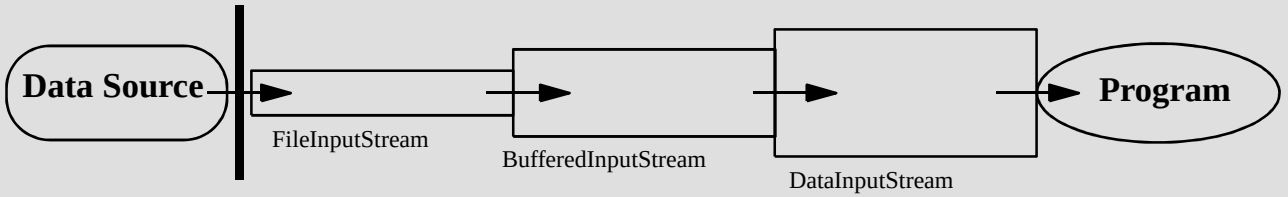
```
1     import java.io.*;
2     public class TestNodeStreams {
3        public static void main(String[] args) {
4           try {
5              FileReader input = new FileReader(args[0]);
6              try {
7                 FileWriter output = new FileWriter(args[1]);
8                 try {
9                    char[] buffer = new char[128];
10                   int charsRead;
11
12                   // read the first buffer
13                   charsRead = input.read(buffer);
14                   while ( charsRead != -1 ) {
15                      // write buffer to the output file
```

A Simple Example

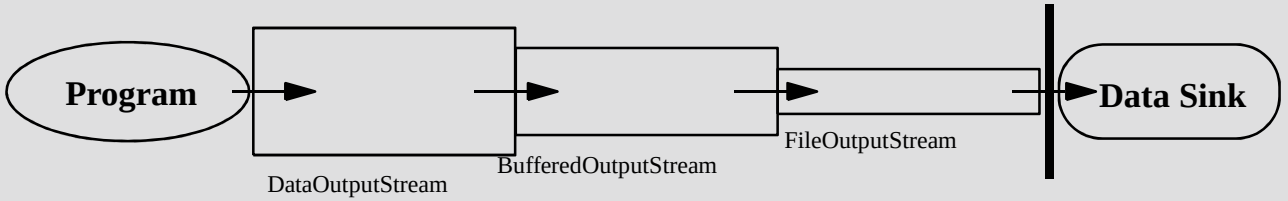
```
16                      output.write(buffer, 0, charsRead);
17
18                      // read the next buffer
19                      charsRead = input.read(buffer);
20                  }
21
22              } finally {
23                  output.close();}
24          } finally {
25              input.close();}
26      } catch (IOException e) {
27          e.printStackTrace();
28      }
29
30  }
}
```

I/O Stream Chaining

Input Stream Chain



Output Stream Chain



Agenda

- Command line arguments
- System properties
- I/O stream fundamentals
- InputStreamReader and OutputStreamWriter
- Object serialization/deserialization
- Console I/O
- java.io.File class
- Java 7 enhancements

Converting streams

- Character oriented streams can be used in conjunction with byte-oriented streams:
  - Use InputStreamReader to "convert" an InputStream to a Reader
  - Use OutputStreamWriter to "convert" an OutputStream to a Writer

- InputStreamReader:
  - This class acts as a bridge from byte streams to character streams
  - InputStreamReader takes an InputStream parameter to its constructor
  - The InputStreamReader reads bytes from the InputStream and translates them into characters according to the specified encoding.
- OutputStreamWriter
  - This class acts as a bridge from character streams to byte streams
  - OutputStreamWriter takes an OutputStream parameter to its constructor
  - Characters written to the OutputStreamWriter are translated to bytes (based on the encoding) and written to the underlying OuputStream.

Converting Streams

- Charset's:
  - UTF8
  - UTF-16
  - Cp1257
  - Cp1251
- More info:

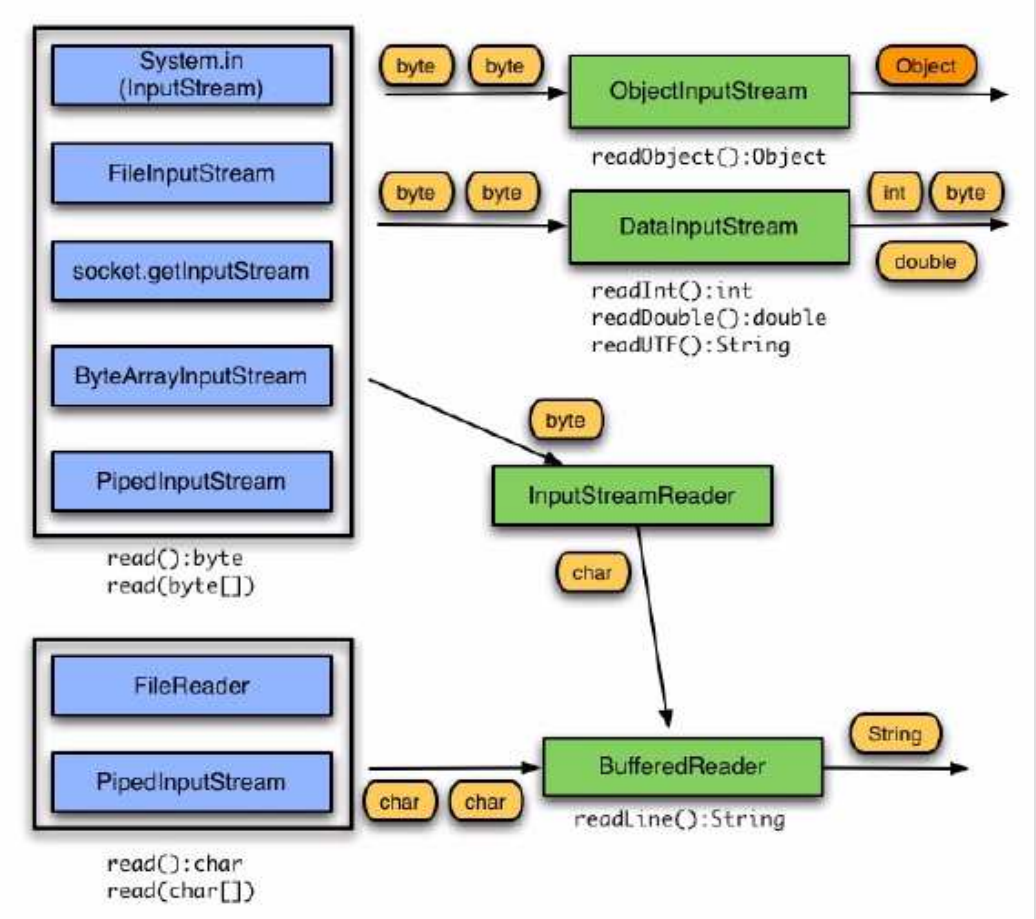
http://docs.oracle.com/javase/1.4.2/docs/guide/intl/encoding.doc.html

Converting Streams

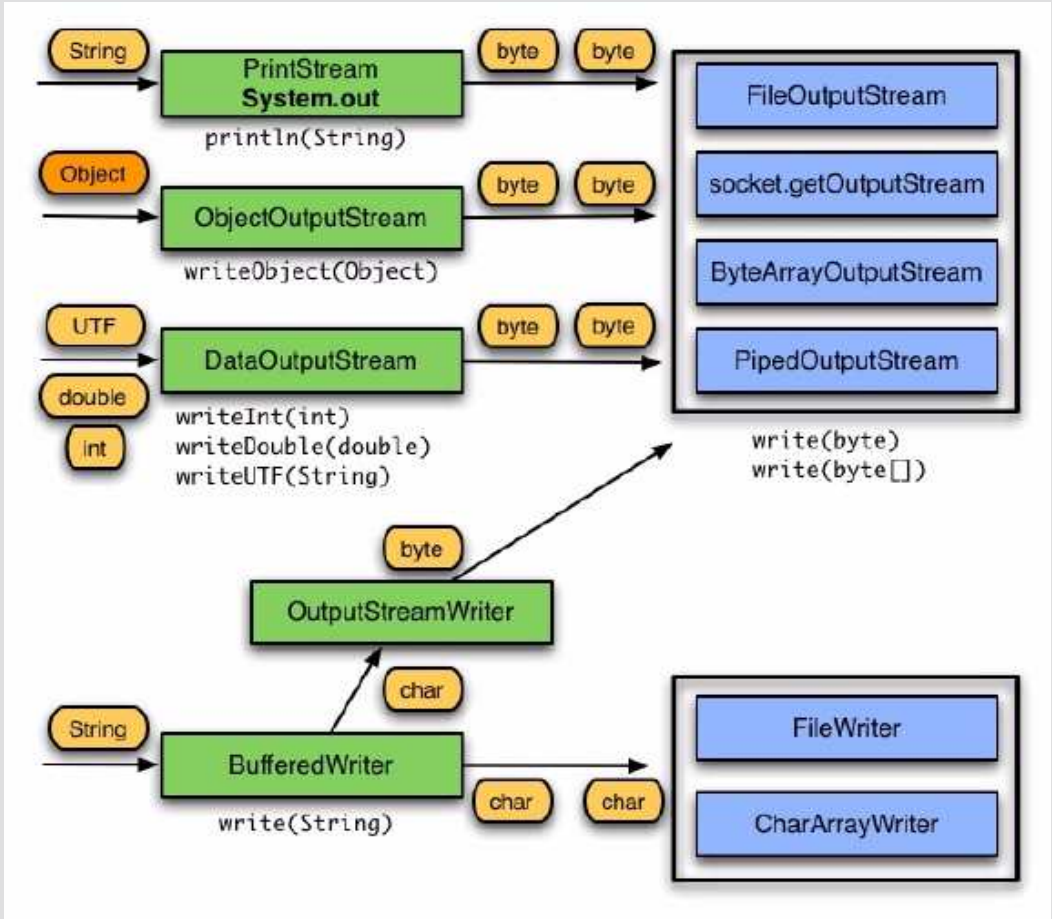
```
FileInputStream fin =
      new FileInputStream("data.txt");
InputStreamReader inr =
      new InputStreamReader(null, "UTF8");
BufferedReader bin =
      new BufferedReader(inr);


FileOutputStream fout =
      new FileOutputStream("output.txt");
OutputStreamWriter outw =
      new OutputStreamWriter(fout, "Cp1257");
BufferedWriter bout =
      new BufferedWriter(outw);
```

Input Chaining Combinations: A Review



Output Chaining Combinations: A Review



Agenda

- Command line arguments
- System properties
- I/O stream fundamentals
- InputStreamReader and OutputStreamWriter
- Object serialization/deserialization
- Console I/O
- java.io.File class
- Java 7 enhancements

Serialization

- Serialization is a mechanism for saving the objects as a sequence of bytes and rebuilding them later when needed.
- When an object is serialized, only the fields of the object are preserved
- When a field references an object, the fields of the referenced object are also serialized
- Some object classes are not serializable because their fields represent transient operating system-specific information.

Serialization

- If an object is to be serialized:
  - The class must be declared as public
  - The class must implement Serializable
  - The class must have a no-argument constructor
  - All fields of the class must be serializable: either primitive types or serializable objects
- Implementing the Serializable interface:
  - The Serializable interface does not define any methods
    - Question: What possible use is there for an interface that does not declare any methods?
    - Answer: Serializable is used as flag to tell Java it needs to do extra work with this class

The SerializeDate Class

```
1   import java.io.*;
2   import java.util.Date;
3
4   public class SerializeDate {
5
6       SerializeDate() {
7           Date d = new Date ();
8
9           try {
10              FileOutputStream f =
11                      new FileOutputStream ("date.ser");
12              ObjectOutputStream s =
13                      new ObjectOutputStream (f);
14              s.writeObject (d);
15              s.close ();
16          } catch (IOException e) {
17              e.printStackTrace ();
18          }
19      }
20
21      public static void main (String args[]) {
22          new SerializeDate();
23      }
24  }
```

The DeSerializeDate Class

```
1   import java.io.*;
2   import java.util.Date;
3
4   public class DeSerializeDate {
5
6       DeSerializeDate () {
7           Date d = null;
8
9           try {
10              FileInputStream f = new FileInputStream ("date.ser");
11              ObjectInputStream s = new ObjectInputStream (f);
12              d = (Date) s.readObject ();
13              s.close ();
14          } catch (Exception e) {
15              e.printStackTrace ();
16          }
17      System.out.println("Deserialized Date object from date.ser");
18      System.out.println("Date: "+d);
19    }
20
21    public static void main (String args[]) {
22      new DeSerializeDate();
23    }
24  }
```

Agenda

- Command line arguments
- System properties
- I/O stream fundamentals
- InputStreamReader and OutputStreamWriter
- Object serialization/deserialization
- Console I/O
- java.io.File class
- Java 7 enhancements

Console I/O

- The variable System.out enables you to write to *standard output*.

  System.out is an object of type PrintStream.
- The variable System.in enables you to read from *standard input*.

  System.in is an object of type InputStream.
- The variable System.err enables you to write to *standard error*.

  System.err is an object of type PrintStream.

Writing to Standard Output

- The `println` methods print the argument and a newline character (`\n`).
- The `print` methods print the argument without a newline character.
- The `print` and `println` methods are overloaded for most primitive types (`boolean`, `char`, `int`, `long`, `float`, and `double`) and for `char[]`, `Object`, and `String`.
- The `print(Object)` and `println(Object)` methods call the `toString` method on the argument.

Reading From Standard Input

```
1   import java.io.*;
2
3   public class KeyboardInput {
4      public static void main (String args[]) {
5         String s;
6         // Create a buffered reader to read
7         // each line from the keyboard.
8         InputStreamReader ir
9            = new InputStreamReader(System.in);
10        BufferedReader in = new BufferedReader(ir);
11
12        System.out.println("Unix: Type ctrl-d to exit." +
13                                    "\nWindows: Type ctrl-z to exit");
```

Reading From Standard Input

```
14          try {
15             // Read each input line and echo it to the screen.
16             s = in.readLine();
17             while ( s != null ) {
18                System.out.println("Read: " + s);
19                s = in.readLine();
20             }
21
22             // Close the buffered reader.
23             in.close();
24          } catch (IOException e) { // Catch any IO exceptions.
25             e.printStackTrace();
26          }
27
28       }
    }
```

Simple Formatted Output

- You can use the formatting functionality as follows:

```
out.printf("name  count\n");
String  s  =  String.format("%s  %5d%n",  user,  total);
```

- Common formatting codes are listed in this table.

Code	Description
%s	Formats the argument as a string, usually by calling the toString method on the object.
%d %o %x	Formats an integer, as a decimal, octal, or hexadecimal value.
%f %g	Formats a floating point number. The %g code uses scientific notation.
%n	Inserts a newline character to the string or stream.
%%	Inserts the % character to the string or stream.

Simple Formatted Input

- The Scanner class provides a formatted input function.
- A Scanner class can be used with console input streams as well as file or network streams.
- You can read console input as follows:

```
1   import java.io.*;
2   import java.util.Scanner;
3   public class ScanTest {
4       public static void main(String [] args) {
5           Scanner s = new Scanner(System.in);
6           String param = s.next();
7           System.out.println("the param 1" + param);
8           int value = s.nextInt();
9           System.out.println("second param" + value);
10          s.close();
11      }
12  }
```

Agenda

- Command line arguments
- System properties
- I/O stream fundamentals
- InputStreamReader and OutputStreamWriter
- Object serialization/deserialization
- Console I/O
- java.io.File class
- Java 7 enhancements

Files and File I/O

The java.io package enables you to do the following:

- Create File objects
- Manipulate File objects
- Read and write to file streams

Creating a New File Object

The File class provides several utilities:

- File myFile;
- myFile = new File("myfile.txt");
- myFile = new File("MyDocs", "myfile.txt");

Directories are treated like files in the Java programming language. You can create a File object that represents a directory and then use it to identify other files, for example:

```
File myDir = new File("MyDocs");
myFile = new File(myDir, "myfile.txt");
```

The File Tests and Utilities

- File information:

```
String  getName()
String  getPath()
String  getAbsolutePath()
String  getParent()
long  lastModified()
long  length()
```

- File modification:

```
boolean  renameTo(File  newName)
boolean  delete()
```

- Directory utilities:

```
boolean  mkdir()
String[]  list()
```

The File Tests and Utilities

- File tests:

```
boolean     exists()
boolean     canWrite()
boolean     canRead()
boolean     isFile()
boolean     isDirectory()
boolean     isAbsolute();
boolean     is  Hidden();
```

File object

- Paths:
- getAbsolutePath() - Returns the absolute pathname string of this abstract pathname
- getCanonicalPath() - Returns the canonical pathname string of this abstract pathname
- absolute vs canonical:
  - is system-dependent, buf if the full path contains aliases, shortcuts, shadows, or symbolic links of some kind, the canonical path resolves those aliases to the actual directories they refer to

File Object: manipulation

- createNewFile() - Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist
- renameTo(File dest) - Renames the file denoted by this abstract pathname. Whether or not this method can move a file from one filesystem to another is platform-dependent
- delete() - Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted
- mkdirs() - Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories
- createTempFile(String prefix, String suffix) - Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name

Copy file

```
FileInputStream in = new FileInputStream(src);
FileOutputStream out = new FileOutputStream(dst);

for (int c = in.read(); c != -1; c = in.read()) {
       out.write(c);
}

in.close();
out.flush();
out.close();
```

File Stream I/O

- For file input:
  - Use the FileReader class to read characters.
  - Use the BufferedReader class to use the readLine method.
- For file output:
  - Use the FileWriter class to write characters.
  - Use the PrintWriter class to use the print and println methods.

File Input Example

A file input example is:

```
1     import java.io.*;
2     public class ReadFile {
3        public static void main (String[] args) {
4           // Create file
5           File file = new File(args[0]);
6
7           try {
8              // Create a buffered reader
9              // to read each line from a file.
10             BufferedReader in
11                = new BufferedReader(new FileReader(file));
12             String s;
13
```

Printing a File

```
14              // Read each line from the file and echo it to the screen.
15              s = in.readLine();
16              while ( s != null ) {
17                  System.out.println("Read: " + s);
18                  s = in.readLine();
19              }
20              // Close the buffered reader
21              in.close();
22
23          } catch (FileNotFoundException e1) {
24              // If this file does not exist
25              System.err.println("File not found: " + file);
26
27          } catch (IOException e2) {
28              // Catch any other IO exceptions.
29              e2.printStackTrace();
30          }
31
32      }
    }
```

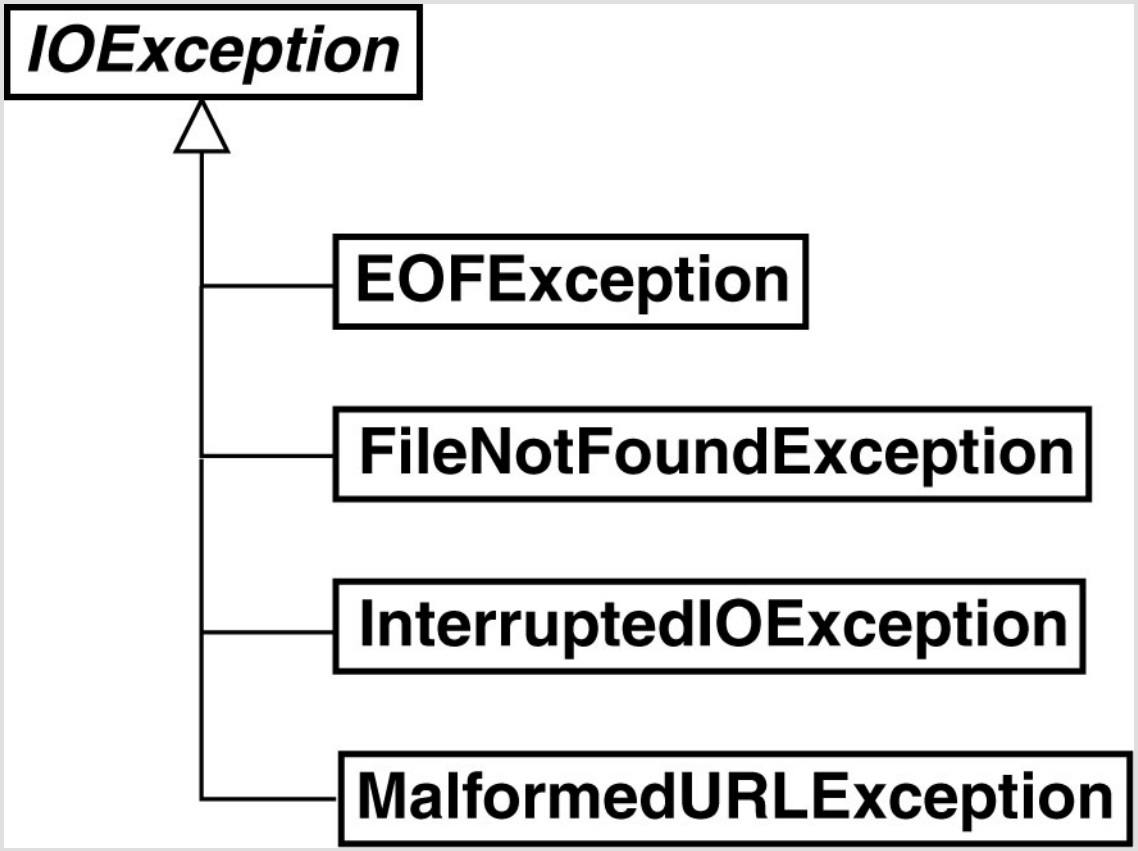
File Output Example

```
1   import java.io.*;
2
3   public class WriteFile {
4      public static void main (String[] args) {
5         // Create file
6         File file = new File(args[0]);
7
8         try {
9            // Create a buffered reader to read each line from standard in.
10           InputStreamReader isr
11              = new InputStreamReader(System.in);
12           BufferedReader in
13              = new BufferedReader(isr);
14           // Create a print writer on this file.
15           PrintWriter out
16              = new PrintWriter(new FileWriter(file));
17           String s;
```

File Output Example

```
18
19              System.out.print("Enter file text.    ");
20              System.out.println("[Type ctrl-d to stop.]");
21
22              // Read each input line and echo it to the screen.
23              while ((s = in.readLine()) != null) {
24                  out.println(s);
25              }
26
27              // Close the buffered reader and the file print writer.
28              in.close();
29              out.close();
30
31          } catch (IOException e) {
32          // Catch any IO exceptions.
33              e.printStackTrace();
34          }
35
36      }
    }
```

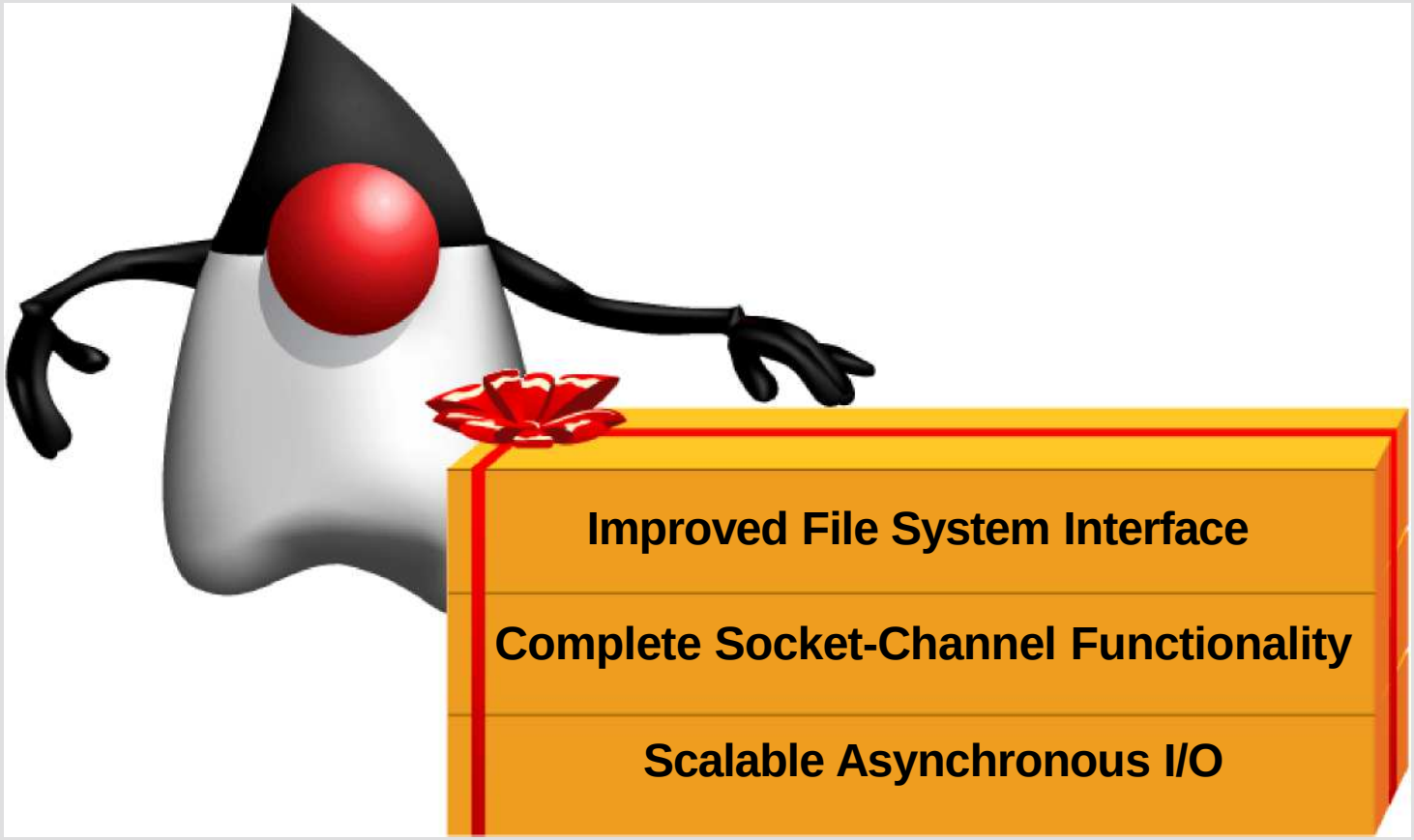
Exceptions



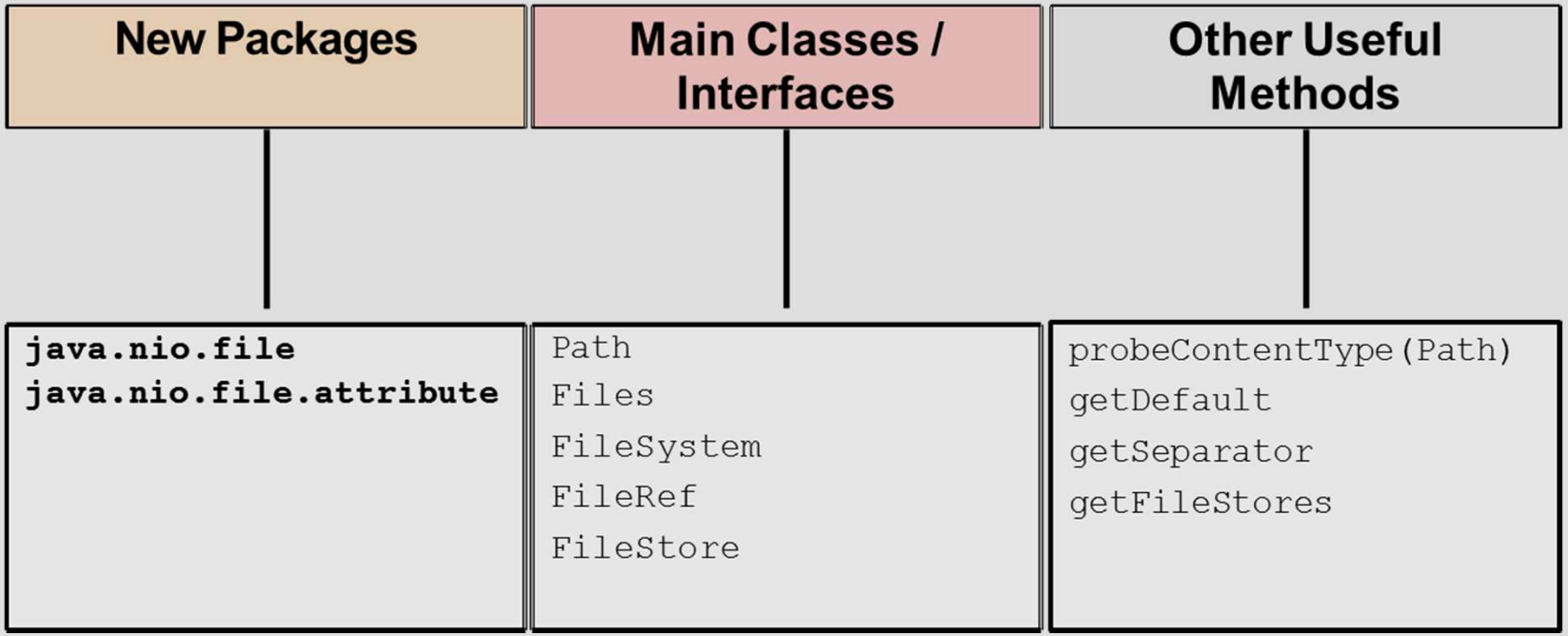
Agenda

- Command line arguments
- System properties
- I/O stream fundamentals
- InputStreamReader and OutputStreamWriter
- Object serialization/deserialization
- Console I/O
- java.io.File class
- Java 7 enhancements

Enhancements in File I/O APIs



The New File System API



Path interface

- `Path` is a programmatic representation of a path in the file system.
- The `Path` interface includes various methods that can be used to obtain:
  - Information about the path
  - Access elements of the path
  - Convert the path to other forms
  - Extract portions of a path and various other operations

File System Access with Path

- Before Java 7:

```
...
File file = new File("fileName"); ...
```

- With Java 7:

```
...
Path path = Paths.get("fileName"); ...
```

- The `File` class has a new method, `toPath()`, that allows you to transform `File` to `Path`.

```
...
Path path = new File("fileName").toPath(); ...
```

`Path` Operations

- Creating a Path
- Retrieving Information About a Path
- Removing Redundancies from a Path
- Converting a Path
- Joining Two Paths
- Creating a Path Between Two Paths
- Comparing Two Paths

Retrieving Information About a `Path`

- Internally, `Path` stores name elements as a sequence.
- The highest element in the directory structure would be located at index `0`.
- The lowest element in the directory structure would be located at index [`n-1`].
- Some important methods include:
  - `getFileName()`
  - `getName(int index)`
  - `getNameCount()`
  - `subpath(int beginIndex, int endIndex)`
  - `getParent()`
  - `getRoot()`

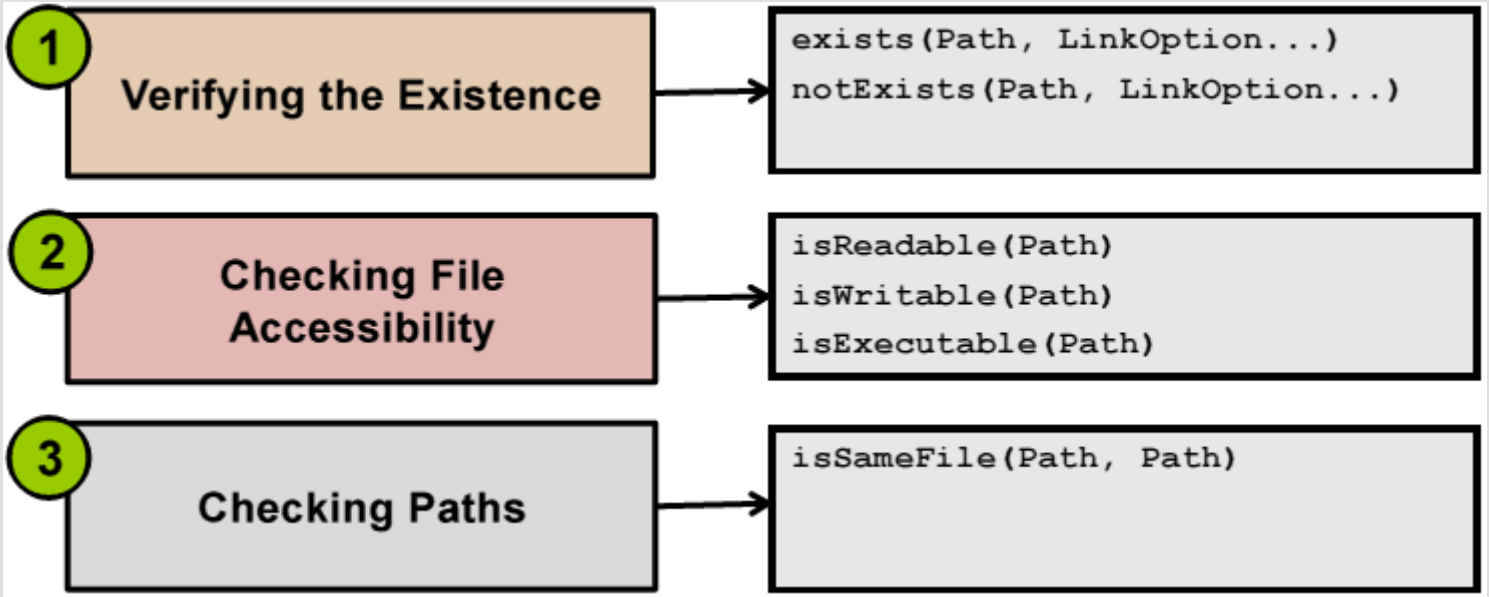
File Operations

- Checking a File or Directory
- Deleting a File or Directory
- Copying a File or Directory
- Moving a File or Directory
- Managing Metadata
- Reading, Writing, and Creating Files
- Random Access Files
- Creating and Reading Directories

- All done by java.nio.file.Files class

Checking a File or Directory

- Does that file exist on the file system?
- Is it readable?
- Is it writable?
- Is it executable?



Deleting a File or Directory

- You can delete files, directories, or links. The Files class provides two methods:
  - delete(Path)
    - Throws NoSuchFileException, DirectoryNotEmptyException or IOExceptioin
  - deleteIfExists(Path)
    - Throws no exceptions

Copying a File or Directory

- You can copy a file or directory by using the copy(Path, Path, CopyOption...) method.
- When directories are copied, the files inside the directory are not copied.
- `java.nio.file.StandardCopyOption`
  - REPLACE_EXISTING
  - COPY_ATTRIBUTES
  - NOFOLLOW_LINKS
- `Files` class also defines methods that may be used to copy between a file and a stream

Moving a File or Directory

- You can move a file or directory by using the `move(Path, Path, CopyOption...)` method
  - REPLACE_EXISTING
  - ATOMIC_MOVE
- Moving a directory will not move the contents of the directory.

Managing Metadata

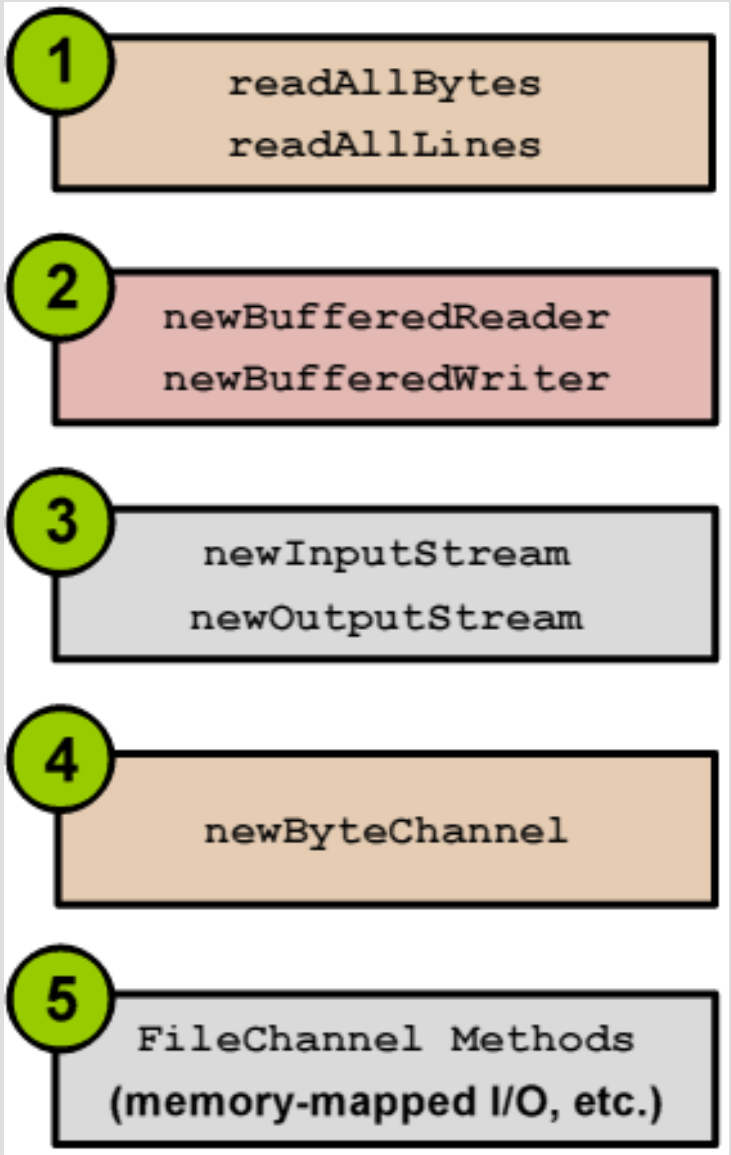
- Metadata is "data about other data."
- Metadata tracks information about each of the file or directory in the file system.
- A file system's metadata is typically referred to as its file attributes.
- The Files class includes methods that can be used to obtain a single attribute of a file, or to set an attribute

```
• size(Path)
• isDirectory(Path, LinkOption)
• isRegularFile(Path, LinkOption...)
• isSymbolicLink(Path)
• isHidden(Path)
• getLastModifiedTime(Path, LinkOption...)
• setLastModifiedTime(Path, FileTime)
```

Managing Metadata

Method	Explanation
`size`	Returns the size of the specified file in bytes
`isDirectory`	Returns true if the specified Path locates a file that is a directory
`isRegularFile`	Returns true if the specified Path locates a file that is a regular file
`isSymbolicLink`	Returns true if the specified Path locates a file that is a symbolic link
`isHidden`	Returns true if the specified Path locates a file that is considered hidden by the file system
`getLastModifiedTime`	Returns or sets the specified file's last modified time
`setLastModifiedTime`	
`getAttribute`	Returns or sets the value of a file attribute
`setAttribute`	

Reading, Writing, and Creating Files



Reading/Writing All Bytes or Lines from a File

- The `readAllBytes` or `readAllLines` method reads entire contents of the file in one pass.
- Example:

```
Path file = ...;
byte[] fileArray;
fileArray = Files.readAllBytes(file);
```

- Use `write` method(s) to write bytes, or lines, to a file.
- Example:

```
Path file = ...;
byte[] buf = ...;
Files.write(file, buf);
```

Watching Directory for Changes

- The java.nio.file package provides a file change notification API, called the WatchService API.
- This API enables you to register a directory (or directories) with the watch service.
- When registering, you tell the service which of the following types of events you are interested in:
  - File creation
  - File deletion
  - File modification