Wrapper Classes

- *Character, Integer, Long, Float,* and *Double*
- class names begin with an upper case letter
- they literally wrap the primitive data type in a class
- are used to provide constants and general methods for the primitive data types.
- Integer.**parse**Int(String s)

Stream Classes

- stream → refers to any input source or output destination for data

input of data → *public* ***BufferedReader****(Reader in);*

- InputStreamReader stream = new InputStreamReader (System.in);
  BufferedReader keyboard = new BufferedReader(stream)
- BufferedReader keyboard = new
  BufferedReader(new InputStreamReader(System.in));
- *keyboard.readline()*
- *int intNumber = new Integer(keyboard.readline()).intValue();*

output of data → *public* ***PrintWriter****(OutputStream out, boolean autoFlush);*

- You are advised to set the *autoflash* argument to *true*, otherwise, you may not get any information to appear on the screen of a monitor
- *PrintWriter screen = new PrintWriter(System.out, true);*
- *screen.println("abc");*
- *screen.print("abc"); screen.flush();*

Without the *flush* method, the output will not be displayed on the screen until the next *println* statement is executed by the computer.

FileReader

```
FileReader file-name1 = new FileReader("pathname");
BufferedReader inputFile = new BufferedReader(file-name1);
…
float-name = new Float(inputFile.readLine()).floatValue();
string-name = inputFile.readLine();
...
inputFile.close();
```

- the double backslash \\ in path name is necessary to avoid any confusion with an escape character in the string.
- *FileNotFoundException*

FileWriter

```
FileWriter file-name2 = new FileWriter("pathname");
PrintWriter outputFile = new PrintWriter(file-name2);
…
outputFile.println();
…
outputFile.close();
```

```
File f  = new File("FileName.txt");
FileInputStream fis = new FileInputStream(f);
BufferedReader input = new BufferedReader(fis);
```

Writing text files

- FileOutputStream outputFile = new
  FileOutputStream("FileName.txt");
  OutputStreamWriter outputStream = new
  OutputStreamWriter(outputFile);
  PrintWriter printWriter = new PrintWriter(outputStream, true);
- FileOutputStream outputFile = new
  FileOutputStream("FileName.txt");
  PrintWriter printWriter = new PrintWriter(outputFile, true);
- FileWriter fileWriter = new FileWriter("info.txt");
  PrintWriter printWriter = new PrintWriter(fileWriter, true);

Reading Text Files

- FileInputStream inputFile = new
  FileInputStream("FileName.txt");
  InputStreamReader reader = new InputStreamReader(inputFile);
- FileReader fileReader = new FileReader("info.txt");

```
import java.io.*;
public class Class1
{
	public static void main(String args[]) throws
FileNotFoundException, IOException
	{

		File f = new File("x.txt");

		FileReader fr = new FileReader(f);
		BufferedReader br = new BufferedReader(fr);

		FileWriter fw = new FileWriter(f);
		PrintWriter pw = new PrintWriter(fw,true);

		pw.println("haha");
		System.out.println(br.readLine());
	}
}
```

StringTokenizer

```
public class StringTokenizer implements Enumeration
```

- public StringTokenizer(String str,String delim, boolean returnTokens);
- public StringTokenizer(String str, String delim);
- **public StringTokenizer(String str);**
- public boolean hasMoreTokens();
- **public String nextToken();**
  return a token delimited by any white space character.
- **public String nextToken (String delim);**
  return a token specially delimited by the character ***delim***.
  - o note that it requires a string argument and not a character argument
- public boolean hasMoreElements();
- public Object nextElement();

- `public int countTokens();`
- If the token delimiter is not specified, then the delimiter is assumed to be any white space character.

java.util.**Random**

Random()

- **nextInt()** → Returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence

java.lang.**Math**

- abs(double)
- abs(float)
- abs(int)
- abs(long)

Class

public final class Class extends Object implements Serializable
java.lang.Class

- Instances of the class Class represent classes and interfaces in a running Java application.
- Every array also belongs to a class that is reflected as a Class object that is shared by all arrays with the same element type and number of dimensions.
- The primitive Java types (boolean, byte, char, short, int, long, float, and double), and the keyword void are also represented as Class objects.
- Class has no public constructor.
- Class objects are constructed automatically by the Java Virtual Machine as classes are loaded and by calls to the defineClass method in the class loader.

method

- static Class **forName**(String className) throws ClassNotFoundException → Returns the Class object associated with the class or interface with the given string name.
- public Constructor[] **getConstructors()** throws SecurityException → Returns an array containing Constructor objects reflecting all the public constructors of the class represented by this Class object. An array of length 0 is returned if the class has no public constructors, or if the class is an array class, or if the class reflects a primitive type or void.
- public String **getName()** → Returns the fully-qualified name of the entity (class, interface, array class, primitive type, or void) represented by this Class object, as a String.

To print the class name of an object:

```
void printClassName(Object obj)
{
    System.out.println("The class of " + obj +
                       " is " + obj.getClass().getName());
}
```

Interface Serializable

- public interface Serializable
- java.io
- Serializability of a class is enabled by the class implementing the java.io.Serializable interface.
- Classes that do not implement this interface will not have any of their state serialized or deserialized.
- All subtypes of a serializable class are themselves serializable.
- The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.
- ObjectOutputStream and ObjectInputStream can provide an application with persistent storage for graphs of objects when used with a FileOutputStream and FileInputStream respectively.
- Classes control how they are serialized by implementing either the java.io.Serializable or java.io.Externalizable interfaces.
- Implementing the Serializable interface allows object serialization to save and restore the entire state of the object and it allows classes to evolve between the time the stream is written and the time it is read.

ObjectOutputStream

- An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream.
- The objects can be read (reconstituted) using an ObjectInputStream.
- Persistent storage of objects can be accomplished by using a file for the stream.
- If the stream is a network socket stream, the objects can be reconsituted on another host or in another process.
- Only objects that support the java.io.Serializable interface can be written to streams.
- The method writeObject is used to write an object to the stream.
- Any object, including Strings and arrays, is written with writeObject.
- Multiple objects or primitives can be written to the stream.
- The objects must be read back from the corresponding ObjectInputstream with the same types and in the same order as they were written.
- Primitive data types can also be written to the stream using the appropriate methods from DataOutput. Strings can also be written using the writeUTF method.

To write an object that can be read by the example in ObjectInputStream:

```
FileOutputStream ostream = new FileOutputStream("t.tmp");
ObjectOutputStream p = new ObjectOutputStream(ostream);
p.writeInt(12345);
p.writeObject("Today");
p.writeObject(new Date());
p.flush();
ostream.close();
```

ObjectInputStream

- An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream.
- is used to recover those objects previously serialized.
- ObjectInputStream ensures that the types of all objects in the graph created from the stream match the classes present in the Java Virtual Machine. Classes are loaded as required using the standard mechanisms.

- Only objects that support the java.io.Serializable or java.io.Externalizable interface can be read from streams.
- The method readObject is used to read an object from the stream. Java's safe casting should be used to get the desired type.
- In Java, strings and arrays are objects and are treated as objects during serialization. When read they need to be cast to the expected type.
- Primitive data types can be read from the stream using the appropriate method on DataInput.
- Reading an object is analogous to running the constructors of a new object. Memory is allocated for the object and initialized to zero (NULL). No-arg constructors are invoked for the non-serializable classes and then the fields of the serializable classes are restored from the stream starting with the serializable class closest to java.lang.object and finishing with the object's most specifiec class.

To read from a stream as written by the example in ObjectOutputStream:

```
FileInputStream istream = new FileInputStream("t.tmp");
ObjectInputStream p = new ObjectInputStream(istream);
int i = p.readInt();
String today = (String)p.readObject();
Date date = (Date)p.readObject();
istream.close();
```

Class Constructor

java.lang.reflect.Constructor

Constructor provides information about, and access to, a single constructor for a class.

Constructor permits widening conversions to occur when matching the actual parameters to newInstance() with the underlying constructor's formal parameters, but throws an IllegalArgumentException if a narrowing conversion would occur.

```
public Object newInstance(Object[] initargs)
                  throws InstantiationException,
                         IllegalAccessException,
                         IllegalArgumentException,
                         InvocationTargetException
```

→ Uses the constructor represented by this Constructor object to create and initialize a new instance of the constructor's declaring class, with the specified initialization parameters.