Chapter 3 API Contents

3.1 Develop code that uses the primitive wrapper classes (such as Boolean, Character, Double, Integer, etc.), and/or autoboxing & unboxing. Discuss the differences between the String, StringBuilder, and StringBuffer classes.

Autoboxing/unboxing of primitive types

Manual conversion between primitive types (such as an `int`) and wrapper classes (such as `Integer`) is necessary when adding a primitive data type to a collection. As an example, consider an `int` being stored and then retrieved from an `ArrayList`:

```
list.add(0, new Integer(59));
int n = ((Integer)(list.get(0))).intValue();
```

The new autoboxing/unboxing feature eliminates this manual conversion. The above segment of code can be written as:

```
list.add(0, 59);
int total = list.get(0);
```

However, note that the wrapper class, `Integer` for example, must be used as a generic type:

```
List<Integer> list = new ArrayList<Integer>();
```

The autoboxing and auto-unboxing of Java primitives produces code that is more concise and easier to follow.

```
int i = 10;

Integer iRef = new Integer(i);  // Explicit Boxing
int j = iRef.intValue();        // Explicit Unboxing

iRef = i;       // Automatic Boxing
j = iRef;       // Automatic Unboxing
```

Automatic boxing and unboxing conversions alleviate the drudgery in converting values of primitive types to objects of the corresponding wrapper classes and vice versa.

Boxing conversion converts primitive values to objects of corresponding wrapper types: if `p` is a value of a `primtiveType`, boxing conversion converts `p` into a reference `ref` of corresponding `WrapperType`, such that `ref.primitiveTypeValue() == p`.

Unboxing conversion converts objects of wrapper types to values of corresponding primitive types: if `ref` is a reference of a `WrapperType`, unboxing conversion converts the reference `ref` into `ref.primitiveTypeValue()`, where `primitiveType` is the primitive type corresponding to the `WrapperType`.

Assignment conversions on `boolean` and numeric types:

```
boolean boolVal = true;
byte b = 2;
short s = 2;
```

```
char c ='2';
int i = 2;

// Boxing
Boolean boolRef = boolVal;
Byte bRef = 88;
Short sRef = 2;
Character cRef = '2';
Integer iRef = 2;
// Integer iRef1 = s; // WRONG ! short not assignable to Integer

// Unboxing
boolean boolVal1 = boolRef;
byte b1 = bRef;
short s1 = sRef;
char c1 = cRef;
int i1 = iRef;
```

Method invocation conversions on actual parameters:

```
...
flipFlop("(String, Integer, int)", new Integer(4), 2004);
...
private static void flipFlop(String str, int i, Integer iRef) {
        out.println(str + " ==> (String, int, Integer)");
}
```

The output:

```
(String, Integer, int) ==> (String, int, Integer)
```

Casting conversions:

```
Integer iRef = (Integer) 2;     // Boxing followed by identity cast
int i = (int) iRef;             // Unboxing followed by identity cast
// Long lRef = 2;                 // WRONG ! Type mismatch: cannot convert from
int to Long
// Long lRef = (Long) 2;        // WRONG ! int not convertible to Long
Long lRef_1 = (long) 2;         // OK ! explicit cast to long
Long lRef_2 = 2L;               // OK ! long literal
```

Numeric promotion: unary and binary:

```
Integer iRef = 2;
long l1 = 2000L + iRef;     // binary numeric promotion
int i = -iRef;              // unary numeric promotion
```

In the `if` statement, condition can be `Boolean`:

```
Boolean expr = true;
if (expr) {
        out.println(expr);
} else {
        out.println(!expr); // Logical complement operator
}
```

In the `switch` statement, the switch expression can be `Character`, `Byte`, `Short` or `Integer`:

```
// Constants
final short ONE = 1;
final short ZERO = 0;
final short NEG_ONE = -1;

// int expr = 1; // (1) short is assignable to int. switch works.
// Integer expr = 1; // (2) short is not assignable to Integer. switch compile
error.
Short expr = 1; // (3) OK. Cast is not required.
switch (expr) { // (4) expr unboxed before case comparison.
        case ONE:
                out.println("ONE"); break;
        case ZERO:
                out.println("ZERO"); break;
        case NEG_ONE:
                out.println("NEG_ONE"); break;
        default:
                assert false;
}
```

The output:

```
ONE
```

In the `while`, `do-while` and `for` statements, the condition can be `Boolean`:

```
Boolean expr = true;
while (expr) {
        expr = !expr;
}

Character[] version = { '5', '.', '0' };    // Assignment: boxing
for (Integer iRef = 0;                      // Assignment: boxing
        iRef < version.length;           // Comparison: unboxing
        ++iRef) {                           // ++: unboxing and boxing
    out.println(iRef + ": " + version[iRef]); // Array index: unboxing
}
```

Output:

```
0: 5
1: .
2: 0
```

Boxing and unboxing in collections/maps:

```
String[] words = new String[] {"aaa", "bbb", "ccc", "aaa"};
Map<String, Integer> m = new TreeMap<String, Integer>();
for (String word : words) {
        Integer freq = m.get(word);
        m.put(word, freq == null ? 1 : freq + 1);
}
out.println(m);
```

The output:

```
{aaa=2, bbb=1, ccc=1}
```

In the next example an `int` is being stored and then retrieved from an `ArrayList`. The J2SE 5.0 leaves the conversion required to transition to an `Integer` and back to the compiler.

Before:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = (list.get(0)).intValue();
```

After:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 42);
int total = list.get(0);
```

An `Integer` expression can have a `null` value. If your program tries to autounbox `null`, it will throw a `NullPointerException`. The `==` operator performs reference identity comparisons on `Integer` expressions and value equality comparisons on `int` expressions. Finally, there are performance costs associated with boxing and unboxing, even if it is done automatically.

Here is another sample program featuring autoboxing and unboxing. It is a static factory that takes an `int` array and returns a `List` of `Integer` backed by the array. This method provides the full richness of the `List` interface atop an `int` array. All changes to the list write through to the array and vice-versa:

```
// List adapter for primitive int array
public static List<Integer> asList(final int[] a) {
        return new AbstractList<Integer>() {

                public Integer get(int i) { return a[i]; }

                // Throws NullPointerException if val == null
                public Integer set(int i, Integer val) {
                        Integer oldVal = a[i];
                        a[i] = val;
                        return oldVal;
                }

                public int size() { return a.length; }
        };
}
```

Wrappers and primitives comparison:

```
public class WrappersTest {
        public static void main(String[] s) {
                Integer i1 = new Integer(2);
                Integer i2 = new Integer(2);
                System.out.println(i1 == i2); // FALSE

                Integer j1 = 2;
                Integer j2 = 2;
                System.out.println(j1 == j2); // TRUE

                Integer k1 = 150;
                Integer k2 = 150;
                System.out.println(k1 == k2); // FALSE

                Integer jj1 = 127;
```

```
                Integer jj2 = 127;
                System.out.println(jj1 == jj2); // TRUE

                int jjj1 = 127;
                Integer jjj2 = 127;
                System.out.println(jjj1 == jjj2); // TRUE

                Integer kk1 = 128;
                Integer kk2 = 128;
                System.out.println(kk1 == kk2); // FALSE

                Integer kkk1 = 128;
                int kkk2 = 128;
                System.out.println(kkk1 == kkk2); // TRUE

                Integer w1 = -128;
                Integer w2 = -128;
                System.out.println(w1 == w2); // TRUE

                Integer m1 = -129;
                Integer m2 = -129;
                System.out.println(m1 == m2); // FALSE

                int mm1 = -129;
                Integer mm2 = -129;
                System.out.println(mm1 == mm2); // TRUE
        }
}
```

NOTE, certain primitives are always to be boxed into the same immutable wrapper objects. These objects are then CACHED and REUSED, with the expectation that these are commonly used objects. These special values are:

- The `boolean` values `true` and `false`.
- The `byte` values.
- The `short` and `int` values between `-128` and `127`.
- The `char` values in the range '`\u0000`' to '`\u007F`'.

```
Character c1 = '\u0000';
Character c2 = '\u0000';
System.out.println("c1 == c2 : " + (c1 == c2)); // TRUE !!!

Character c11 = '\u00FF';
Character c12 = '\u00FF';
System.out.println("c11 == c12 : " + (c11 == c12)); // FALSE

c1 == c2 : true
c11 == c12 : false
```

The following example gives `NullPointerException`:

```
Integer i = null;
int j = i; // java.lang.NullPointerException !!!
```

This example demonstrates methods resolution when selecting overloaded method:

```
public static void main(String[] args) {
        doSomething(1);
        doSomething(2.0);
```

```
}

public static void doSomething(double num) {
        System.out.println("double : " + num);
}

public static void  doSomething(Integer num) {
        System.out.println("Integer : " + num);
}
```

The output is:

```
double : 1.0
double : 2.0
```

Java 5.0 will always select the same method that would have been selected in Java 1.4.

The method resolution inludes three passes:

1. Attempt to locate the correct method WITHOUT any boxing, unboxing, or vararg invocations.
2. Attempt to locate the correct method WITH boxing, unboxing, and WITHOUT any vararg invocations.
3. Attempt to locate the correct method WITH boxing, unboxing, or vararg invocations.

String

The `String` class represents character strings. All string literals in Java programs, such as "`abc`", are implemented as instances of this class.

All implemented interfaces: `Serializable`, `CharSequence`, `Comparable<String>`.

This method returns the number of characters in the string as in:

```
int length ()
```

This example results in variable `x` holding the value `8`:

```
String str = "A string";
int x = str.length ();
```

Removes whitespace from the leading and trailing edges of the string:

```
String trim ()
```

This results in the variable `str` referencing the string "`14 units`":

```
String string = "  14 units  ";
String str = string.trim ();
```

The following methods return the index, starting from `0`, for the location of the given character in the string. (The `char` value will be widened to `int`):

```
int indexOf (int ch)
int lastIndexOf (int ch)
```

For example:

```
String string = "One fine day";
int x = string.indexOf ('f');
```

This results in a value of `4` in the variable `x`. If the string holds NO such character, the method returns `-1`.

To continue searching for more instances of the character, you can use the method:

```
indexOf(int ch, int fromIndex)
```

This will start the search at the `fromIndex` location in the string and search to the end of the string.

The methods:

```
indexOf (String str)
indexOf (String str, int fromIndex)
```

provide similar functions but search for a sub-string rather than just for a single character.

Similarly, the methods:

```
lastIndexOf (int ch)
lastIndexOf (int ch, int fromIndex)

lastIndexOf (String str)
lastIndexOf (String str, int fromIndex)
```

search backwards for characters and strings starting from the right side and moving from right to left. (The `fromIndex` second parameter still counts from the left, with the search continuing from that index position toward the beginning of the string).

These two methods test whether a string begins or ends with a particular substring:

```
boolean startsWith (String prefix)
boolean endsWith (String str)
```

For example:

```
String [] str = {"Abe", "Arthur", "Bob"};
for (int i=0; i < str.length (); i++) {
        if (str1.startsWith ("Ar")) doSomething ();
}
```

The first method return a new string with all the characters set to lower case while the second returns the characters set to upper case:

```
String toLowerCase ()
String toUpperCase ()
```

Example:

```
String [] str = {"Abe", "Arthur", "Bob"};
for (int i=0; i < str.length(); i++){
```

```
    if (str1.toLowerCase ().startsWith ("ar")) doSomething ();
}
```

StringBuilder

J2SE5.0 added the `StringBuilder` class, which is a drop-in replacement for `StringBuffer` in cases where thread safety is not an issue. Because `StringBuilder` is NOT synchronized, it offers FASTER performance than `StringBuffer`.

In general, you should use `StringBuilder` in preference over `StringBuffer`. In fact, the J2SE 5.0 `javac` compiler normally uses `StringBuilder` instead of `StringBuffer` whenever you perform string concatenation as in:

```
System.out.println ("The result is " + result);
```

All the methods available on `StringBuffer` are also available on `StringBuilder`, so it really is a drop-in replacement.

Instances of `StringBuilder` are not safe for use by multiple threads. If such synchronization is required then it is recommended that `StringBuffer` be used.

StringBuffer

String objects are immutable, meaning that once created they cannot be altered. Concatenating two strings does not modify either string but instead creates a new string object:

```
String str = "This is";
str = str + " a new string object";
```

Here `str` variable now references a completely new object that holds the "`This is a new string object`" string.

This is not very efficient if you are doing extensive string manipulation with lots of new strings created through this sort of append operations. The `String` class maintains a pool of strings in memory. String literals are saved there and new strings are added as they are created. Extensive string manipulation with lots of new strings created with the `String` append operations can therefore result in lots of memory taken up by unneeded strings. Note however, that if two string literals are the same, the second string reference will point to the string already in the pool rather than create a duplicate.

The class `java.util.StringBuffer` offers more efficient string creation. For example:

```
StringBuffer strb = new StringBuffer ("This is");
strb.append(" a new string object");
System.out.println (strb.toString());
```

The `StringBuffer` uses an internal `char` array for the intermediate steps so that new strings objects are not created. If it becomes full, the array is copied into a new larger array with the additional space available for more append operations.

The `StringBuffer` is a thread-safe, mutable sequence of characters. A string buffer is like a `String`, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

The principal operations on a `StringBuffer` are the `append` and `insert` methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string buffer. The `append` method always adds these characters at the end of the buffer; the `insert` method adds the characters at a specified point.

For example, if `z` refers to a string buffer object whose current contents are "`start`", then the method call `z.append("le")` would cause the string buffer to contain "`startle`", whereas `z.insert(4, "le")` would alter the string buffer to contain "`starlet`".

In general, if `sb` refers to an instance of a `StringBuffer`, then `sb.append(x)` has the same effect as `sb.insert(sb.length(), x)`.

Whenever an operation occurs involving a source sequence (such as appending or inserting from a source sequence) this class synchronizes only on the string buffer performing the operation, not on the source.

Every string buffer has a capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made larger. As of release JDK 5, this class has been supplemented with an equivalent class designed for use by a single thread, `StringBuilder`. The `StringBuilder` class should generally be used in preference to this one, as it supports all of the same operations but it is faster, as it performs no synchronization.

`StringBuffer` class DOES NOT override the `equals()` method. Therefore, it uses `Object` class' `equals()`, which only checks for equality of the object references. `StringBuffer.equals()` does not return `true` even if the two `StringBuffer` objects have the same contents:

```
StringBuffer sb = new StringBuffer("ssss");
StringBuffer sb_2 = new StringBuffer("ssss");
out.println("sb equals sb_2 : " + sb.equals(sb_2));
```

The output:

```
sb equals sb_2 : false
```

NOTE, `String`'s `equals()` method checks if the argument if of type string, if not it returns `false`:

```
StringBuffer sb = new StringBuffer("ssss");
String st = new String("ssss");
out.println("sb equals st : " + sb.equals(st)); // Always 'false'
out.println("st equals sb : " + st.equals(sb)); // Always 'false'
```

The output:

```
sb equals st : false
st equals sb : false
```

3.2 Given a scenario involving navigating file systems, reading from files, or writing to files, develop the correct solution using the following classes (sometimes in combination), from java.io: BufferedReader, BufferedWriter, File, FileReader, FileWriter and PrintWriter.

File

Files and directories are accessed and manipulated via the `java.io.File` class. The `File` class does not actually provide for input and output to files. It simply provides an identifier of files and directories.

Always remember that just because a `File` object is created, it does not mean there actually exists on the disk a file with the identifier held by that `File` object.

The `File` class includes several overloaded constructors. For example, the following instance of `File` refers to a file named `myfile.txt` in the current directory of the program that the JVM is running:

```
File file = new File ("myfile.txt");
```

Again, the file `myfile.txt` may or may not exist in the file system. An attempt to use `File` object that refers to a file that does not exist will cause a `FileNotFoundException` to be thrown.

The `File` instance can also created with a filename that includes path:

```
File fileA = new File("/tmp/filea.txt");
```

Another overloaded constructor allows separate specification of the path and the file name:

```
File fileA = new File("/tmp", "filea.txt");
```

Directories can also be specified:

```
File directoryA = new File("/tmp");
```

There are a number of useful methods in `File`, e.g.:

```
boolean exist();        // does the file exist
boolean canWrite();     // can the file be written to
boolean canRead();      // can the file be read
boolean isFile();       // does it represent a file
boolean isDirectory();  // or a directory
```

There are also methods to get the file name and path components, to make a directory, to get a listing of the files in a directory, etc.

```
String getName();           // get the name of the file (no path included)
String getPath();           // get the abstract file path
String getCanonicalPath();  // get the name of the file with path
String getAbsolutePath();   // get the absolute file path.
```

NOTE, the `getCanonicalPath()` method first converts this pathname to absolute form if necessary, as if by invoking the `getAbsolutePath()` method, and then maps it to its unique form in a system-dependent way. This typically involves removing redundant names such as "." and ".." from the

pathname, resolving symbolic links (on UNIX platforms), and converting drive letters to a standard case (on Microsoft Windows platforms).

NOTE, that path names use different separator characters on different hosts. Windows uses "\", Unix - "/", Macintosh - ":". The static variables:

```
File.separator         // string with file separator
File.separatorChar     // char with file separator
File.pathSeparator     // string with path separator
File.pathSeparatorChar // char with path separator
```

can be used to insure that your programs are platform independent. For example, this snippet shows how to build a platform independent path:

```
String dirName = "dataDir";
String filename = "data.dat";
File filData = new File(dirName + File.separator + filename);
```

The `File` class includes the method:

```
boolean mkdir();
```

This method will create a directory with the abstract path name represented by the `File` object if that `File` object represents a directory. The method returns `true` if the directory creation succeeds and `false` if not. A situation in which the directory cannot be created is, for example, when the `File` object refers to an actual file that already exists in the file system.

Instances of the `File` class are immutable; that is, once created, the abstract pathname represented by a `File` object will never change.

This example lists the files and subdirectories in a directory:

```
File dir = new File("directoryName");

String[] children = dir.list();
if (children == null) {
        // Either dir does not exist or is not a directory
} else {
        for (int i=0; i<children.length; i++) {
                // Get filename of file or directory
                String filename = children[i];
        }
}
```

It is also possible to filter the list of returned files. This example does not return any files that start with '.' :

```
FilenameFilter filter = new FilenameFilter() {
        public boolean accept(File dir, String name) {
                return !name.startsWith(".");
        }
};
children = dir.list(filter);
```

The list of files can also be retrieved as `File` objects:

```
File[] files = dir.listFiles();
```

This filter only returns directories:

```
FileFilter fileFilter = new FileFilter() {
        public boolean accept(File file) {
                return file.isDirectory();
        }
};
files = dir.listFiles(fileFilter);
```

FileReader

Convenience class for reading character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an `InputStreamReader` on a `FileInputStream`.

`FileReader` is meant for reading streams of characters. For reading streams of raw bytes, consider using a `FileInputStream`.

Constructors:

```
public FileReader(String fileName) throws FileNotFoundException
public FileReader(File file) throws FileNotFoundException
public FileReader(FileDescriptor fd)
```

Some methods:

```
// Read a single character
public int read() throws IOException

// Read characters into a portion of an array
public int read(char cbuf[], int offset, int length) throws IOException

// Read characters into an array.  This method will block until some input
// is available, an I/O error occurs, or the end of the stream is reached.
public int read(char cbuf[]) throws IOException

try {
        BufferedReader in = new BufferedReader(new FileReader("inFileName"));
        String str;
        while ((str = in.readLine()) != null) {
                process(str);
        }
        in.close();
} catch (IOException e) {
}
```

BufferedReader

Read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

In general, each `read` request made of a `Reader` causes a corresponding `read` request to be made of the underlying character or byte stream. It is therefore advisable to wrap a `BufferedReader` around any `Reader` whose `read()` operations may be costly, such as `FileReaders` and `InputStreamReaders`. For example:

```
BufferedReader in = new BufferedReader(new FileReader("foo.in"));
```

will buffer the input from the specified file. Without buffering, each invocation of `read()` or `readLine()` could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

Programs that use `DataInputStreams` for textual input can be localized by replacing each `DataInputStream` with an appropriate `BufferedReader`.

Constructors:

```
public BufferedReader(Reader in, int sz)
public BufferedReader(Reader in)
```

Some methods:

```
// Read a single character
public int read() throws IOException

// Read characters into a portion of an array
public int read(char cbuf[], int off, int len) throws IOException

// Read a line of text.  A line is considered to be terminated by any one
// of a line feed ('\n'), a carriage return ('\r'), or a carriage return
// followed immediately by a linefeed.
//
// ignoreLF - If true, the next '\n' will be skipped
String readLine(boolean ignoreLF) throws IOException

// Read a line of text.  A line is considered to be terminated by any one
// of a line feed ('\n'), a carriage return ('\r'), or a carriage return
// followed immediately by a linefeed.
// same as 'readLine(false)'
public String readLine() throws IOException

// Methods to work with marks (positions in the stream)
public void mark(int readAheadLimit)  throws IOException
public void reset()  throws IOException
public boolean markSupported()

// Example reads from from one file and writes to another.
// Assume inFile and outFile are Strings with path/file names.

try {
        BufferedReader bufReader = new BufferedReader(new FileReader(inFile));
        BufferedWriter bufWriter = new BufferedWriter(new FileWriter(outFile));

        String line = null;
        while ((line=bufReader.readLine()) != null){
                bufWriter.write(line);
                bufWriter.newLine();   // adds newline character(s)
        }

        bufReader.close();
        bufWriter.close();
} catch (IOException e) {
}
```

FileWriter

Convenience class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an `OutputStreamWriter` on a `FileOutputStream`.

Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one `FileWriter` (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.

`FileWriter` is meant for writing streams of characters. For writing streams of raw bytes, consider using a `FileOutputStream`.

Constructors:

```
public FileWriter(String fileName) throws IOException

// if 'append' parameter is 'true', then data will be written
// to the end of the file rather than the beginning
public FileWriter(String fileName, boolean append) throws IOException

public FileWriter(File file) throws IOException

public FileWriter(File file, boolean append) throws IOException

public FileWriter(FileDescriptor fd)
```

Some methods:

```
// Write an array of characters
public void write(char[] cbuf) throws IOException

// Write a single character
public void write(int c) throws IOException

// Write a portion of an array of characters
public void write(char cbuf[], int off, int len) throws IOException

// Write a string
public void write(String str) throws IOException

// Write a portion of a string
public void write(String str, int off, int len) throws IOException
```

Writing to a `File`. If the file does not already exist, it is automatically created:

```
try {
        BufferedWriter out = new BufferedWriter(new FileWriter("outFileName"));
        out.write("aString");
        out.close();
} catch (IOException e) {
}
```

Appending to a `File`:

```
try {
        BufferedWriter  out  =  new  BufferedWriter(new  FileWriter("fileName",
true));
        out.write("aString");
```

```
        out.close();
} catch (IOException e) {
}
```

BufferedWriter

Write text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.

The buffer size may be specified, or the default size may be accepted. The default is large enough for most purposes.

A `newLine()` method is provided, which uses the platform's own notion of line separator as defined by the system property `line.separator`. Not all platforms use the newline character ('\n') to terminate lines. Calling this method to terminate each output line is therefore preferred to writing a newline character directly.

In general, a `Writer` sends its output immediately to the underlying character or byte stream. Unless prompt output is required, it is advisable to wrap a `BufferedWriter` around any `Writer` whose `write()` operations may be costly, such as `FileWriters` and `OutputStreamWriters`. For example:

```
PrintWriter      out      =      new      PrintWriter(new      BufferedWriter(new
FileWriter("foo.out")));
```

will buffer the `PrintWriter`'s output to the file. Without buffering, each invocation of a `print()` method would cause characters to be converted into bytes that would then be written immediately to the file, which can be very inefficient.

Constructors:

```
public BufferedWriter(Writer out)
public BufferedWriter(Writer out, int sz) // sz – Output-buffer size
```

Some methods:

```
// Write a portion of an array of characters
public void write(char cbuf[], int off, int len) throws IOException

// Write a single character
public void write(int c) throws IOException

// Write a portion of a string
public void write(String str, int off, int len) throws IOException

// Write an array of characters       (inherited from class java.io.Writer)
public void write(char[] cbuf) throws IOException

// Write a string (inherited from class java.io.Writer)
public void write(String str) throws IOException

// Write a line separator. The line separator string is defined by the system
property
// 'line.separator', and is not necessarily a single newline ('\n') character.
public void newLine() throws IOException

// Flush the stream
public void flush() throws IOException

try {
```

```
        BufferedWriter bw = new BufferedWriter(new FileWriter("aaa.txt"));
        bw.write ("Java");
        bw.flush ();
        bw.close ();
} catch (IOException e) {
}
```

PrintWriter

Print formatted representations of objects to a text-output stream. This class implements all of the `print` methods found in `PrintStream`. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

Unlike the `PrintStream` class, if automatic flushing is enabled it will be done only when one of the `println`, `printf`, or `format` methods is invoked, rather than whenever a newline character happens to be output. These methods use the platform's own notion of line separator rather than the newline character.

Methods in this class never throw I/O exceptions, although some of its constructors may. The client may inquire as to whether any errors have occurred by invoking `checkError()`.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class FileOutput {
    public static void main(String[] args) {
        try {
            BufferedWriter out = new BufferedWriter(new FileWriter("test.txt"));
            PrintWriter pw = new PrintWriter(out);
            int i = 30;
            double j = 102.90;
            boolean b = true;
            pw.print(i);
            pw.print(j);
            pw.print(b);
            pw.close();
            out.close();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

The `test.txt` file contents:

```
30102.9true
```

3.3 Develop code that serializes and/or de-serializes objects using the following APIs from java.io: DataInputStream, DataOutputStream, FileInputStream, FileOutputStream, ObjectInputStream, ObjectOutputStream and Serializable.

DataInputStream

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.

A `DataInputStream` takes as a constructor an `InputStream`.

Some methods:

```
public final boolean readBoolean() throws IOException
public final byte readByte() throws IOException
public final int readUnsignedByte() throws IOException
public final short readShort() throws IOException
public final int readUnsignedShort() throws IOException
public final char readChar() throws IOException
public final int readInt() throws IOException
public final long readLong() throws IOException
public final float readFloat() throws IOException
public final double readDouble() throws IOException

FileInputStream fis = new FileInputStream("test.txt");
DataInputStream dis = new DataInputStream(fis);
boolean value = dis.readBoolean();
```

DataOutputStream

A data output stream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

A `DataOutputStream` takes as a constructor an `OutputStream`.

Some methods:

```
public final void writeBoolean(boolean v) throws IOException
public final void writeByte(int v) throws IOException
public final void writeShort(int v) throws IOException
public final void writeChar(int v) throws IOException
public final void writeInt(int v) throws IOException
public final void writeLong(long v) throws IOException
public final void writeFloat(float v) throws IOException
public final void writeDouble(double v) throws IOException
public final void writeBytes(String s) throws IOException
public final void writeChars(String s) throws IOException
```

FileInputStream

A `FileInputStream` obtains input bytes from a file in a file system. What files are available depends on the host environment.

`FileInputStream` is meant for reading streams of raw bytes such as image data. For reading streams of characters, consider using `FileReader`.

Some methods:

```
// Reads a byte of data from this input stream
public int read() throws IOException
public int read(byte[] b) throws IOException
public int read(byte[] b, int off, int len) throws IOException

FileInputStream fis = new FileInputStream("test.txt");
```

FileOutputStream

A file output stream is an output stream for writing data to a `File` or to a `FileDescriptor`. Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one `FileOutputStream` (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.

`FileOutputStream` is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using `FileWriter`.

Some methods:

```
// Writes the specified byte to this file output stream
public void write(int b) throws IOException
public void write(byte[] b) throws IOException
public void write(byte[] b, int off, int len) throws IOException

FileOutputStream fos = new FileOutputStream("testout.txt");

// append
FileOutputStream fos = new FileOutputStream("testout.txt", true);
```

ObjectInputStream

An `ObjectInputStream` deserializes primitive data and objects previously written using an `ObjectOutputStream`. `ObjectOutputStream` and `ObjectInputStream` can provide an application with persistent storage for graphs of objects when used with a `FileOutputStream` and `FileInputStream` respectively. `ObjectInputStream` is used to recover those objects previously serialized. Other uses include passing objects between hosts using a socket stream or for marshaling and unmarshaling arguments and parameters in a remote communication system.

`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`.

The default deserialization mechanism for objects restores the contents of each field to the value and type it had when it was written. Fields declared as `transient` or `static` are IGNORED by the deserialization process. References to other objects cause those objects to be read from the stream as necessary. Graphs of objects are restored correctly using a reference sharing mechanism. New objects are always allocated when deserializing, which prevents existing objects from being overwritten.

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 specific class:

```
public class A {
    public int aaa = 111;

    public A() {
        System.out.println("A");
    }
}
```

Class `B` extends non-serializable class `A`:

```
import java.io.Serializable;

public class B extends A implements Serializable {

    public int bbb = 222;

    public B() {
        System.out.println("B");
    }
}
```

The client code:

```
public class Client {
    public static void main(String[] args) throws Exception {

        B b = new B();
        b.aaa = 888;
        b.bbb = 999;

        System.out.println("Before serialization:");
        System.out.println("aaa = " + b.aaa);
        System.out.println("bbb = " + b.bbb);

        ObjectOutputStream save = new ObjectOutputStream(
                                    new FileOutputStream("datafile"));
        save.writeObject(b); // Save object
        save.flush(); // Empty output buffer

        ObjectInputStream restore = new ObjectInputStream(
                                    new FileInputStream("datafile"));
        B z = (B) restore.readObject();

        System.out.println("After deserialization:");
        System.out.println("aaa = " + z.aaa);
        System.out.println("bbb = " + z.bbb);
    }
}
```

The client's output:

```
A
B
Before serialization:
aaa = 888
bbb = 999
```

```
A
After deserialization:
aaa = 111
bbb = 999
```

For example to read from a stream as written by the example in `ObjectOutputStream`:

```
FileInputStream fis = new FileInputStream("test.tmp");
ObjectInputStream ois = new ObjectInputStream(fis);

int i = ois.readInt();
String today = (String) ois.readObject();
Date date = (Date) ois.readObject();

ois.close();
```

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. It automatically traverses references between objects, saving and restoring entire graphs.

Serializable classes that require special handling during the serialization and deserialization process should implement the following methods:

```
private void writeObject(java.io.ObjectOutputStream stream) throws IOException;
private void readObject(java.io.ObjectInputStream stream) throws IOException,
        ClassNotFoundException;
```

The `readObject` method is responsible for reading and restoring the state of the object for its particular class using data written to the stream by the corresponding `writeObject` method. The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is restored by reading data from the `ObjectInputStream` for the individual fields and making assignments to the appropriate fields of the object. Reading primitive data types is supported by `DataInput`.

Serialization does not read or assign values to the fields of any object that does not implement the `java.io.Serializable` interface. Subclasses of Objects that are not serializable can be serializable. In this case the non-serializable class must have a no-arg constructor to allow its fields to be initialized. In this case it is the responsibility of the subclass to save and restore the state of the non-serializable class. It is frequently the case that the fields of that class are accessible (`public`, package, or `protected`) or that there are get and set methods that can be used to restore the state.

Implementing the `Externalizable` interface allows the object to assume complete control over the contents and format of the object's serialized form. The methods of the `Externalizable` interface, `writeExternal` and `readExternal`, are called to save and restore the objects state. When implemented by a class they can write and read their own state using all of the methods of `ObjectOutput` and `ObjectInput`. It is the responsibility of the objects to handle any versioning that occurs.

Enum constants are deserialized differently than ordinary serializable or externalizable objects. The serialized form of an `enum` constant consists solely of its name; field values of the constant are not transmitted. To deserialize an `enum` constant, `ObjectInputStream` reads the constant name from the stream; the deserialized constant is then obtained by calling the static method `Enum.valueOf(Class, String)` with the `enum` constant's base type and the received constant name as arguments. Like other serializable or externalizable objects, `enum` constants can function as the targets of back references appearing subsequently in the serialization stream. The process by which `enum` constants are deserialized

CANNOT be customized: any class-specific `readObject`, `readObjectNoData`, and `readResolve` methods defined by enum types are ignored during deserialization.

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 reconstituted on another host or in another process.

Only objects that support the `java.io.Serializable` interface can be written to streams. The class of each serializable object is encoded including the class name and signature of the class, the values of the object's fields and arrays, and the closure of any other objects referenced from the initial objects.

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.

The default serialization mechanism for an object writes the class of the object, the class signature, and the values of all non-transient and non-static fields. References to other objects (except in `transient` or `static` fields) cause those objects to be written also. Multiple references to a single object are encoded using a reference sharing mechanism so that graphs of objects can be restored to the same shape as when the original was written.

For example to write an object that can be read by the example in `ObjectInputStream`:

```
FileOutputStream fos = new FileOutputStream("test.tmp");
ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeInt(12345);
oos.writeObject("Today");
oos.writeObject(new Date());

oos.close();
```

Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

```
private void readObject(java.io.ObjectInputStream stream)
        throws IOException, ClassNotFoundException;
private void writeObject(java.io.ObjectOutputStream stream)
        throws IOException
```

The `writeObject` method is responsible for writing the state of the object for its particular class so that the corresponding `readObject` method can restore it. The method does not need to concern itself with the state belonging to the object's superclasses or subclasses. State is saved by writing the individual fields to the `ObjectOutputStream` using the `writeObject` method or by using the methods for primitive data types supported by `DataOutput`.

Serialization does not write out the fields of any object that does not implement the `java.io.Serializable` interface. Subclasses of Objects that are not serializable can be serializable. In this case the non-serializable class must have a no-arg constructor to allow its fields to be initialized. In this case it is the responsibility of the subclass to save and restore the state of the non-serializable class. It is frequently the case that the fields of that class are accessible (`public`, package, or `protected`) or that there are get and set methods that can be used to restore the state.

Serializable

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.

To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's `public`, `protected`, and (if accessible) package fields. The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state. It is an error to declare a class `Serializable` if this is not the case. The error will be detected at runtime.

During deserialization, the fields of non-serializable classes will be initialized using the `public` or `protected` no-arg constructor of the class. A no-arg constructor must be accessible to the subclass that is serializable. The fields of serializable subclasses will be restored from the stream.

When traversing a graph, an object may be encountered that does not support the `Serializable` interface. In this case the `NotSerializableException` will be thrown and will identify the class of the non-serializable object.

Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException
private void readObject(java.io.ObjectInputStream in) throws IOException,
       ClassNotFoundException;
```

Serialization

Imagine a graph of objects that lead from the object to be saved. The entire graph must be saved and restored:

```
Obj 1 --> Obj 2 --> Obj 3
      \--> Obj 4 --> Obj 5
                  \ --> Obj 6
```

We need a byte-coded representation of objects that can be stored in a file external to Java programs, so that the file can be read later and the objects can be reconstructed. Serialization provides a mechanism for saving and restoring objects.

Serializing an object means to code it as an ordered series of bytes in such a way that it can be rebuilt (really a copy) from that byte stream. Deserialization generates a new live object graph out of the byte stream.

The serialization mechanism needs to store enough information so that the original object can be recreated including all objects to which it refers (the object graph).

Java has classes (in the `java.io` package) that allow the creation of streams for object serialization and methods that write to and read from these streams.

Only an object of a class that implements the EMPTY interface `java.io.Serializable` or a subclass of such a class can be serialized.

What is saved:

- The class of the object.
- The class signature of the object.

- All instance variables NOT declared `transient`.
- Objects referred to by non-transient instance variables.

If a duplicate object occurs when traversing the graph of references, only ONE copy is saved, but references are coded so that the duplicate links can be restored.

Saving an object (an array of `Fruit`)

1. Open a file and create an `ObjectOutputStream` object.

```
ObjectOutputStream save =
            new ObjectOutputStream(new FileOutputStream("datafile"));
```

2. Make `Fruit` serializable:

```
class Fruit implements Serializable
```

3. Write an object to the stream using `writeObject()`.

```
Fruit [] fa = new Fruit[3];
// Create a set of 3 Fruits and place them in the array.
...
save.writeObject(fa); // Save object (the array)
save.flush(); // Empty output buffer
```

Restoring the object

1. Open a file and create an `ObjectInputStream` object.

```
ObjectInputStream restore =
            new ObjectInputStream(new FileInputStream("datafile"));
```

2. Read the object from the stream using `readObject()` and then cast it to its appropriate type.

```
Fruit[] newFa;
// Restore the object:
newFa = (Fruit[])restore.readObject();
```

or

```
Object ob = restore.readObject();
```

When an object is retrieved from a stream, it is validated to ensure that it can be rebuilt as the intended object. Validation may fail if the class definition of the object has changed.

A class whose objects are to be saved must implement interface `Serializable`, with no methods, or the `Externalizable` interface, with two methods. Otherwise, runtime exception will be thrown:

```
Exception in thread "main" java.io.NotSerializableException: Bag
       at java.io.ObjectOutputStream.writeObject0(Unknown Source)
       at java.io.ObjectOutputStream.writeObject(Unknown Source)
       at Client.main(Client.java:17)
```

The first superclass of the class (maybe `Object`) that is not serializable must have a no-parameter constructor.

The class must be visible at the point of serialization.

The `implements Serializable` clause acts as a tag indicating the possibility of serializing the objects of the class.

All primitive types are serializable.

Transient fields (with `transient` modifier) are NOT serialized, (i.e., not saved or restored).

A class that implements `Serializable` must mark `transient` fields of classes that do not support serialization (e.g., a file stream).

Because the deserialization process will create new instances of the objects. Comparisons based on the "==" operator MAY NO longer be valid.

Main saving objects methods:

```
public ObjectOutputStream(OutputStream out) throws IOException
public final void writeObject(Object obj) throws IOException
public void flush() throws IOException
public void close() throws IOException
```

Main restoring objects methods:

```
public ObjectInputStream(InputStream in) throws IOException, SecurityException
public final Object readObject() throws IOException, ClassNotFoundException
public void close() throws IOException
```

`ObjectOutputStream` and `ObjectInputStream` also implement the methods for writing and reading primitive data and `Strings` from the interfaces `DataOutput` and `DataInput`, for example:

```
writeBoolean(boolean b)    <==> boolean readBoolean()
writeChar(char c)          <==> char readChar()
writeInt(int i)            <==> int readInt()
writeDouble(double d)      <==> double readDouble()
```

No methods or class variables are saved when an object is serialized.

A class knows which methods and static data are defined in it.

Serialization example (client class):

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Client {
    public static void main(String ... aaa) throws IOException,
                                                    ClassNotFoundException {
        Bag b = new Bag();
        Bag a = null;
```

```
        ObjectOutputStream save = new ObjectOutputStream(
                                        new FileOutputStream("datafile"));
        System.out.println("Before serialization:");
        System.out.println(b);
        save.writeObject(b); // Save object
        save.flush(); // Empty output buffer

        ObjectInputStream restore = new ObjectInputStream(
                                        new FileInputStream("datafile"));
        a = (Bag) restore.readObject();
        System.out.println("After deserialization:");
        System.out.println(a);
    }
}
```

If the `Bag` class does not implement `Serializable`:

```
import java.util.Arrays;

public class Bag {

        Fruit[] fruits = new Fruit[3];

        public Bag() {
                fruits[0] = new Fruit("Orange");
                fruits[1] = new Fruit("Apple");
                fruits[2] = new Fruit("Pear");
        }

        public String toString() {
                return "Bag of fruits :" + Arrays.toString(fruits);
        }
}
```

We get the following runtime exception:

```
Before serialization:
Bag of fruits :[Fruit : Orange, Fruit : Apple, Fruit : Pear]
Exception in thread "main" java.io.NotSerializableException: Bag
        at java.io.ObjectOutputStream.writeObject0(Unknown Source)
        at java.io.ObjectOutputStream.writeObject(Unknown Source)
        at Client.main(Client.java:17)
```

Now, make `Bag` class implement `Serializable`, but `Fruit` class still is not `Serializable`:

```
public class Fruit {

        String name = "";

        public Fruit(String name) {
                this.name = name;
        }

        public String toString() {
                return "Fruit : " + name;
        }
}
```

We get same exception but in class `Fruit`:

```
Before serialization:
Bag of fruits :[Fruit : Orange, Fruit : Apple, Fruit : Pear]
Exception in thread "main" java.io.NotSerializableException: Fruit
        at java.io.ObjectOutputStream.writeObject0(Unknown Source)
        at java.io.ObjectOutputStream.writeArray(Unknown Source)
        at java.io.ObjectOutputStream.writeObject0(Unknown Source)
        ...
```

We can ask not to serialize `Fruit` classes by making array of fruits `transient`:

```
import java.io.Serializable;
import java.util.Arrays;

public class Bag implements Serializable {

        transient Fruit[] fruits = new Fruit[3]; // do not save to disk

        public Bag() {
                fruits[0] = new Fruit("Orange");
                fruits[1] = new Fruit("Apple");
                fruits[2] = new Fruit("Pear");
        }

        public String toString() {
                return "Bag of fruits :" + Arrays.toString(fruits);
        }
}
```

Now the program is running without exceptions, but fruits are not restoring in the bag after deserialization:

```
Before serialization:
Bag of fruits :[Fruit : Orange, Fruit : Apple, Fruit : Pear]
After deserialization:
Bag of fruits :null
```

Only when both `Bag` and `Fruits` are serializable, we get expected output:

```
import java.io.Serializable;
import java.util.Arrays;

public class Bag implements Serializable {

        Fruit[] fruits = new Fruit[3];

        public Bag() {
                fruits[0] = new Fruit("Orange");
                fruits[1] = new Fruit("Apple");
                fruits[2] = new Fruit("Pear");
        }

        public String toString() {
                return "Bag of fruits :" + Arrays.toString(fruits);
        }
}

import java.io.Serializable;

public class Fruit implements Serializable {
```

```
        String name = "";

        public Fruit(String name) {
                this.name = name;
        }

        public String toString() {
                return "Fruit : " + name;
        }
}
```

The client's output now will be:

```
Before serialization:
Bag of fruits :[Fruit : Orange, Fruit : Apple, Fruit : Pear]
After deserialization:
Bag of fruits :[Fruit : Orange, Fruit : Apple, Fruit : Pear]
```

NOTE, `static` and `transient` fields are NOT serialized:

```
import java.io.Serializable;

public class MyClass implements Serializable {
        transient int one;
        private int two;
        static int three;

        public MyClass() {
                one = 1;
                two = 2;
                three = 3;
        }

        public String toString() {
                return "one : " + one + ", two: " + two + ", three: " + three;
        }
}
```

This code saves class instance:

```
...
MyClass b = new MyClass();
ObjectOutputStream        save        =        new        ObjectOutputStream(new
FileOutputStream("datafile"));
save.writeObject(b); // Save object
save.flush(); // Empty output buffer
...
```

Deserialize:

```
...
MyClass a = null;
ObjectInputStream restore = new ObjectInputStream(
                                                new FileInputStream("datafile"));
a = (MyClass) restore.readObject();
System.out.println("After deserialization:");
System.out.println(a);
...
```

The output:

```
After deserialization:
one : 0, two: 2, three: 0
```

As you can see, `static` and `transient` fields got the default values for instance variables.

3.4 Use standard J2SE APIs in the java.text package to correctly format or parse dates, numbers, and currency values for a specific locale; and, given a scenario, determine the appropriate methods to use if you want to use the default locale or a specific locale. Describe the purpose and use of the java.util.Locale class.

Formatting and parsing a `Date` using default formats

Every locale has four default formats for formatting and parsing dates. They are called `SHORT`, `MEDIUM`, `LONG`, and `FULL`. The `SHORT` format consists entirely of numbers while the `FULL` format contains most of the date components. There is also a default format called `DEFAULT` and is the same as `MEDIUM`.

This example formats dates using the default locale (which is "`ru_RU`"). If the example is run in a different locale, the output will not be the same:

```
import static java.lang.System.out;
import java.text.DateFormat;
import java.text.ParseException;
import java.util.Date;
import java.util.Locale;

public class DateFormatExample1 {

    public static void main(String[] args) {
        // Format
        Date date = new Date();

        String s = DateFormat.getDateInstance(DateFormat.SHORT).format(date);
        out.println("DateFormat.SHORT : " + s);

        s = DateFormat.getDateInstance(DateFormat.MEDIUM).format(date);
        out.println("DateFormat.MEDIUM : " + s);

        s = DateFormat.getDateInstance(DateFormat.LONG).format(date);
        out.println("DateFormat.LONG : " + s);

        s = DateFormat.getDateInstance(DateFormat.FULL).format(date);
        out.println("DateFormat.FULL : " + s);

        s = DateFormat.getDateInstance().format(date);
        out.println("default : " + s);

        s = DateFormat.getDateInstance(DateFormat.DEFAULT).format(date);
        out.println("DateFormat.DEFAULT : " + s);

        // Parse
        try {
            date = DateFormat.getDateInstance(DateFormat.DEFAULT).
                                                    parse("08.02.2005");
            out.println("Parsed date : " + date);
        } catch (ParseException e) {
            out.println(e);
        }
    }
}
```

The output:

```
DateFormat.SHORT : 08.02.05
DateFormat.MEDIUM : 08.02.2005
```

```
DateFormat.LONG : 8 Февраль 2005 г.
DateFormat.FULL : 8 Февраль 2005 г.
default : 08.02.2005
DateFormat.DEFAULT : 08.02.2005
Parsed date : Tue Feb 08 00:00:00 EET 2005
```

`DateFormat` is an `abstract` class for date/time formatting subclasses which formats and parses dates or time in a language-independent manner. The date/time formatting subclass, such as `SimpleDateFormat`, allows for formatting (i.e., date -> text), parsing (text -> date), and normalization. The date is represented as a `Date` object or as the milliseconds since January 1, 1970, 00:00:00 GMT.

`DateFormat` helps you to format and parse dates for any locale. Your code can be completely independent of the locale conventions for months, days of the week, or even the calendar format: lunar vs. solar.

To format a date for the current `Locale`, use one of the `static` factory methods:

```
myString = DateFormat.getDateInstance().format(myDate);
```

To format a date for a different `Locale`, specify it in the call to `getDateInstance()`:

```
DateFormat df = DateFormat.getDateInstance(DateFormat.LONG, Locale.US);
```

You can use a `DateFormat` to parse also:

```
myDate = df.parse(myString);
```

NOTE, the `parse(...)` method throws `ParseException` if the beginning of the specified string cannot be parsed. The `ParseException` is a checked exception, so parsing always should be done inside `try-catch` block.

Formatting and parsing a date for a `Locale`

To format and parse in a particular locale, specify the locale when creating the `DateFormat` object:

```
import static java.lang.System.out;
import java.text.DateFormat;
import java.text.ParseException;
import java.util.Date;
import java.util.Locale;

public class DateFormatExample2 {

    public static void main(String[] args) {
        //  Format
        Date date = new Date();
        Locale locale = Locale.US;

        String s = DateFormat.getDateInstance(DateFormat.SHORT,
                                                locale).format(date);
        out.println("DateFormat.SHORT : " + s);

        s = DateFormat.getDateInstance(DateFormat.MEDIUM, locale).format(date);
        out.println("DateFormat.MEDIUM : " + s);

        s = DateFormat.getDateInstance(DateFormat.LONG, locale).format(date);
        out.println("DateFormat.LONG : " + s);
```

```
        s = DateFormat.getDateInstance(DateFormat.FULL, locale).format(date);
        out.println("DateFormat.FULL : " + s);

        s = DateFormat.getDateInstance(DateFormat.DEFAULT, locale).format(date);
        out.println("DateFormat.DEFAULT : " + s);

        // Parse
        try {
            date = DateFormat.getDateInstance(DateFormat.DEFAULT,
                                              locale).parse("Feb 8, 2005");
            out.println("Parsed date : " + date);
        } catch (ParseException e) {
            out.println(e);
        }
    }
}
```

The output will be:

```
DateFormat.SHORT : 2/8/05
DateFormat.MEDIUM : Feb 8, 2005
DateFormat.LONG : February 8, 2005
DateFormat.FULL : Tuesday, February 8, 2005
DateFormat.DEFAULT : Feb 8, 2005
Parsed date : Tue Feb 08 00:00:00 EET 2005
```

Formatting and parsing a number using a default format

`NumberFormat` is the `abstract` base class for all number formats. This class provides the interface for formatting and parsing numbers. `NumberFormat` also provides methods for determining which locales have number formats, and what their names are.

`NumberFormat` helps you to format and parse numbers for any locale. Your code can be completely independent of the locale conventions for decimal points, thousands-separators, or even the particular decimal digits used, or whether the number format is even decimal.

To format a number for the current `Locale`, use one of the factory class methods:

```
myString = NumberFormat.getInstance().format(myNumber);
```

To format a number for a different `Locale`, specify it in the call to `getInstance()`:

```
NumberFormat nf = NumberFormat.getInstance(Locale.US);
```

Use `getInstance()` or `getNumberInstance()` to get the normal number format. Use `getIntegerInstance()` to get an integer number format. Use `getCurrencyInstance` to get the currency number format. And use `getPercentInstance` to get a format for displaying percentages. With this format, a fraction like 0.53 is displayed as 53%.

```
import static java.lang.System.out;
import java.text.DateFormat;
import java.text.NumberFormat;
import java.text.ParseException;

public class NumberFormatExample1 {

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

```
        // Format
        long iii = 1000;
        Long jjj = 1000L;
        NumberFormat format = NumberFormat.getInstance();
        out.println("Format long : " + format.format(iii));
        out.println("Format Long : " + format.format(jjj));

        // Parse
        try {
            Number nnn1 = format.parse("1000");
            out.println("Parsed Number 1  : " + nnn1);

            Number nnn2 = format.parse("1 000");
            out.println("Parsed Number 2  : " + nnn2);
        } catch (ParseException e) {
            out.println(e);
        }
    }
}
```

The output is:

```
Format long : 1 000
Format Long : 1 000
Parsed Number 1  : 1000
Parsed Number 2  : 1
```

Formatting and parsing a number for a `Locale`

A formatted number consists of locale-specific symbols such as the decimal point. This example demonstrates how to format a number for a particular locale:

```
import static java.lang.System.out;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

public class NumberFormatExample2 {

    public static void main(String[] args) {
        // Format
        double iii = 1000.123;
        Double jjj = 1000.123;

        NumberFormat formatUS = NumberFormat.getInstance(Locale.US);
        out.println("Format double (US): " + formatUS.format(iii));
        out.println("Format Double (US): " + formatUS.format(jjj));

        NumberFormat formatDE = NumberFormat.getInstance(Locale.GERMANY);
        out.println("Format double (DE): " + formatDE.format(iii));
        out.println("Format Double (DE): " + formatDE.format(jjj));

        // Parse
               try {
            Number nnn1 = formatUS.parse("1234.1234");
            out.println("Parsed Number 1 (US) : " + nnn1);

            Number nnn2 = formatUS.parse("1234,1234");
            out.println("Parsed Number 2 (US) : " + nnn2);

            Number nnn3 = formatDE.parse("1234.1234");
```

```
            out.println("Parsed Number 3 (DE) : " + nnn3);

            Number nnn4 = formatDE.parse("1234,1234");
            out.println("Parsed Number 4 (DE) : " + nnn4);
        } catch (ParseException e) {
            out.println(e);
        }
    }
}
```

The output will be:

```
Format double (US): 1,000.123
Format Double (US): 1,000.123
Format double (DE): 1.000,123
Format Double (DE): 1.000,123
Parsed Number 1 (US) : 1234.1234
Parsed Number 2 (US) : 12341234
Parsed Number 3 (DE) : 12341234
Parsed Number 4 (DE) : 1234.1234
```

Formatting and parsing currency

The `NumberFormat` provides 2 factory methods for currency format instances:

```
public static final NumberFormat getCurrencyInstance()
public static NumberFormat getCurrencyInstance(Locale inLocale)

import static java.lang.System.out;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

public class CurrencyFormatExample {

    public static void main(String[] args) {
        // Format
        double iii = 999.99;

        NumberFormat format = NumberFormat.getCurrencyInstance();
        NumberFormat formatUS = NumberFormat.getCurrencyInstance(Locale.US);
        NumberFormat formatDE = NumberFormat.
                                    getCurrencyInstance(Locale.GERMANY);

        out.println("Format currency (default): " + format.format(iii));
        out.println("Format currency (US): " + formatUS.format(iii));
        out.println("Format currency (DE): " + formatDE.format(iii));

        // Parse
        try {
            Number nnn1 = format.parse("1234,12 руб.");
            out.println("Parsed currency 1 : " + nnn1);

            Number nnn2 = formatUS.parse("$1234.12");
            out.println("Parsed currency 2 (US) : " + nnn2);

            Number nnn3 = formatDE.parse("1234,12 €");
            out.println("Parsed currency 3 (DE) : " + nnn3);

        } catch (ParseException e) {
            out.println(e);
```

```
        }
    }
}
```

The output will be similar to the following:

```
Format currency (default): 999,99 руб.
Format currency (US): $999.99
Format currency (DE): 999,99 €
Parsed currency 1 : 1234.12
Parsed currency 2 (US) : 1234.12
Parsed currency 3 (DE) : 1234.12
```

`java.util.Locale` class

A `Locale` object represents a specific geographical, political, or cultural region. An operation that requires a `Locale` to perform its task is called locale-sensitive and uses the `Locale` to tailor information for the user. For example, displaying a number is a locale-sensitive operation - the number should be formatted according to the customs/conventions of the user's native country, region, or culture.

Create a `Locale` object using the constructors in this class:

```
Locale(String language)
Locale(String language, String country)
Locale(String language, String country, String variant)
```

The `Locale` class provides a number of convenient constants that you can use to create `Locale` objects for commonly used locales. For example, the following creates a `Locale` object for the United States:

```
Locale locale = Locale.US;
```

You can get the default locale using `static Locale.getDefault()` method:

```
Locale locale = Locale.getDefault();
```

The Java 2 platform provides a number of classes that perform locale-sensitive operations. For example, the `NumberFormat` class formats numbers, currency, or percentages in a locale-sensitive manner. Classes such as `NumberFormat` have a number of convenience methods for creating a default object of that type. For example, the `NumberFormat` class provides these three convenience methods for creating a default `NumberFormat` object:

```
// default locale
NumberFormat.getInstance()
NumberFormat.getCurrencyInstance()
NumberFormat.getPercentInstance()
```

These methods have two variants; one with an explicit locale and one without; the latter using the default locale.

```
// custom locale
NumberFormat.getInstance(myLocale)
NumberFormat.getCurrencyInstance(myLocale)
NumberFormat.getPercentInstance(myLocale)
```

A `Locale` is the mechanism for identifying the kind of object (`NumberFormat`) that you would like to get. The locale is just a mechanism for identifying objects, not a container for the objects themselves.

3.5 Write code that uses standard J2SE APIs in the java.util and java.util.regex packages to format or parse strings or streams. For strings, write code that uses the Pattern and Matcher classes and the String.split(...) method. Recognize and use regular expression patterns for matching (limited to: . (dot), * (star), + (plus), ?, \d, \s, \w, [], ()). The use of *, +, and ? will be limited to greedy quantifiers, and the parenthesis operator will only be used as a grouping mechanism, not for capturing content during matching. For streams, write code using the Formatter and Scanner classes and the PrintWriter.format/printf methods. Recognize and use formatting parameters (limited to: %b, %c, %d, %f, %s) in format strings.

The Java 2 Platform, Standard Edition (J2SE), version 1.4, contains a new package called `java.util.regex`, enabling the use of regular expressions. Now functionality includes the use of meta characters, which gives regular expressions versatility.

A regular expression, specified as a string, must first be compiled into an instance of this class. The resulting pattern can then be used to create a `Matcher` object that can match arbitrary character sequences against the regular expression. All of the state involved in performing a match resides in the matcher, so many matchers can share the same pattern.

A typical invocation sequence is thus:

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

A `matches` method is defined by `Pattern` class as a convenience for when a regular expression is used just once. This method compiles an expression and matches an input sequence against it in a single invocation. The statement:

```
boolean b = Pattern.matches("a*b", "aaaaab");
```

is equivalent to the three statements above, though for repeated matches it is less efficient since it does not allow the compiled pattern to be reused.

Instances of `Pattern` class are immutable and are safe for use by multiple concurrent threads. Instances of the `Matcher` class are not safe for such use.

Character classes

```
[abc]           a, b, or c (simple class)
[^abc]          Any character except a, b, or c (negation)
[a-zA-Z]        a through z or A through Z, inclusive (range)
[a-d[m-p]]      a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]    d, e, or f (intersection)
[a-z&&[^bc]]    a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]   a through z, and not m through p: [a-lq-z](subtraction)
```

Predefined character classes

```
.       Any character (may or may not match line terminators)
\d      A digit: [0-9]
\D      A non-digit: [^0-9]
\s      A whitespace character: [ \t\n\x0B\f\r]
\S      A non-whitespace character: [^\s]
\w      A word character: [a-zA-Z_0-9]
```

```
\W      A non-word character: [^\w]
```

Pattern Class

An instance of the `Pattern` class represents a regular expression that is specified in string form in a syntax similar to that used by Perl.

A regular expression, specified as a string, must first be compiled into an instance of the `Pattern` class. The resulting pattern is used to create a `Matcher` object that matches arbitrary character sequences against the regular expression. Many matchers can share the same pattern because it is stateless.

The `compile` method compiles the given regular expression into a pattern, then the `matcher` method creates a matcher that will match the given input against this pattern. The `pattern` method returns the regular expression from which this pattern was compiled.

The `split` method is a convenience method that splits the given input sequence around matches of this pattern. The following example uses `split` to break up a string of input separated by commas and/or whitespace:

```
import java.util.regex.*;

public class Splitter {
        public static void main(String[] args) throws Exception {
                // Create a pattern to match breaks
                Pattern p = Pattern.compile("[,\\s]+");
                // Split input with the pattern
                String[] result = p.split("one,two, three   four ,  five");
                for (int i=0; i<result.length; i++) {
                        System.out.println("|" + result[i] + "|");
                }
        }
}
```

The output:

```
|one|
|two|
|three|
|four|
|five|
```

Matcher Class

Instances of the `Matcher` class are used to match character sequences against a given string sequence pattern. Input is provided to matchers using the `CharSequence` interface to support matching against characters from a wide variety of input sources.

A matcher is created from a pattern by invoking the pattern's `matcher` method. Once created, a matcher can be used to perform three different kinds of match operations:

- The `matches` method attempts to match the entire input sequence against the pattern.
- The `lookingAt` method attempts to match the input sequence, starting at the beginning, against the pattern.
- The `find` method scans the input sequence looking for the next sequence that matches the pattern.

Each of these methods returns a `boolean` indicating success or failure. More information about a successful match can be obtained by querying the state of the matcher.

The `Matcher` class also defines methods for replacing matched sequences by new strings whose contents can, if desired, be computed from the match result.

The `appendReplacement` method appends everything up to the next match and the replacement for that match. The `appendTail` appends the strings at the end, after the last match.

The following code samples demonstrate the use of the `java.util.regex` package. This code writes "`One dog, two dogs in the yard`" to the standard-output stream:

```
import java.util.regex.*;

public class Replacement {
        public static void main(String[] args) throws Exception {
                // Create a pattern to match cat
                Pattern p = Pattern.compile("cat");
                // Create a matcher with an input string
                Matcher m = p.matcher("One cat, two cats in the yard");
                StringBuffer sb = new StringBuffer();
                boolean result = m.find();
                // Loop through and create a new String with the replacements
                while(result) {
                        m.appendReplacement(sb, "dog");
                        result = m.find();
                }
                // Add the last segment of input to the new String
                m.appendTail(sb);
                System.out.println(sb.toString());
        }
}
```

Quantifiers

Quantifiers specify the number of occurrences of a pattern. This allows us to control how many times a pattern occurs in a string. Table summarizes how to use quantifiers:

Table 3.1. Quantifiers

Greedy Quantifiers	Reluctant Quantifiers	Possessive Quantifiers	Occurrence of a pattern X
X?	X??	X?+	X, once or not at all
X*	X*?	X*+	X, zero or more times
X+	X+?	X++	X, one or more times
X{n}	X{n}?	X{n}+	X, exactly n times
X{n,}	X{n,}?	X{n,}+	X, at least n times
X{n,m}	X{n,m}?	X{n,m}+	X, at least n but not more than m times

The first three columns show regular expressions that represent a set of strings in which `X` loops occur. The last column describes the meaning of its corresponding regular expressions. There are three types of quantifiers to specify each kind of pattern occurrence. These three types of quantifiers are different in usage. It's important to understand the meaning of the metacharacters used in quantifiers before we explain the differences.

The most general quantifier is `{n,m}`, where `n` and `m` are integers. `X{n,m}` means a set of strings in which `X` loops at least `n` times but no more than `m` times. For instance, `X{3, 5}` includes `XXX`, `XXXX`, and `XXXXX` but excludes `X`, `XX`, and `XXXXXX`.

Even though we have the above metacharacters to control occurrence, there are several other ways to match a string with a regular expression. This is why there is a greedy quantifier, reluctant quantifier, and possessive quantifier in each case of occurrence.

A greedy quantifier forces a `Matcher` to digest the whole inputted string first. If the matching fails, it then forces the `Matcher` to back off the inputted string by one character, check matching, and repeat the process until there are no more characters left.

A reluctant quantifier, on the other hand, asks a `Matcher` to digest the first character of the whole inputted string first. If the matching fails, it appends its successive character and checks again. It repeats the process until the `Matcher` digests the whole inputted string.

A possessive quantifier, unlike the other two, makes a `Matcher` digest the whole string and then stop.

Table below helps to understand the difference between the greedy quantifier (the first test), the reluctant quantifier (the second test), and the possessive quantifier (the third test). The string content is "`whellowwwwwwhellowwwwww`"

Table 3.2. Difference between quantifiers

Regular Expression	Result
.*hello	Found the text "whellowwwwwwhello" starting at index 0 and ending at index 17.
.*?hello	Found the text "whello" starting at index 0 and ending at index 6. Found the text "wwwwwwhello" starting at index 6 and ending at index 17.
.*+hello	No match found.

Capturing groups

The above operations also work on groups of characters by using capturing groups. A capturing group is a way to treat a group of characters as a single unit. For instance, `(java)` is a capturing group, where `java` is a unit of characters. `javajava` can belong to a regular expression of `(java)*`. A part of the inputted string that matches a capturing group will be saved and then recalled by back references.

Java provides numbering to identify capturing groups in a regular expression. They are numbered by counting their opening parentheses from left to right. For example, there are four following capturing groups in the regular expression `((A)(B(C)))`:

1. `((A)(B(C)))`
2. `(A)`
3. `(B(C))`
4. `(C)`

You can invoke the `Matcher` method `groupCount()` to determine how many capturing groups there are in a `Matcher`'s `Pattern`.

The numbering of capturing groups is necessary to recall a stored part of a string by back references. A back reference is invoked by `\n`, where `n` is the index of a subgroup to recall the capturing group.

Table 3.3. Groups usage

Whole Content	Regular Expression	Result
abab	([a-z][a-z])\1	Found the text "abab" starting at index 0 and ending at index 4.
abcd	([a-z][a-z])\1	No match found.
abcd	([a-z][a-z])	Found the text "ab" starting at index 0 and ending at index 2. I found the text "cd" starting at index 2 and ending at index 4.

String.split()

J2SE 1.4 added the `split()` method to the `String` class to simplify the task of breaking a string into substrings, or tokens. This method uses the concept of a regular expression to specify the delimiters. A regular expression is a remnant from the Unix `grep` tool ("grep" meaning "general regular expression parser").

See most any introductory Unix text or the Java API documentation for the `java.util.regex.Pattern` class.

In its simplest form, searching for a regular expression consisting of a single character finds a match of that character. For example, the character '`x`' is a match for the regular expression "`x`".

The `split()` method takes a parameter giving the regular expression to use as a delimiter and returns a `String` array containing the tokens so delimited. Using `split()` function:

```
String str = "This is a string object";
String[] words = str.split (" ");
for (String word : words) {
        out.println (word);
}
```

The output:

```
This
is
a
string
object
```

NOTE, `str.split (" ");` is equal to `str.split ("\\s");`.

To use "`*`" (which is a "special" regex character) as a delimiter, specify "`\\*`" as the regular expression (escape it):

```
String str = "A*bunch*of*stars";
String[] starwords = str.split ("\\*");

A
bunch
of
stars
```

NOTE, always use double "`\`" for escaping in java source code, i.e. "`\\s`", "`\\d`", "`\\*`", otherwise the code will not compile:

```
String str = "boo and foo";
str.split("\s"); // WRONG ! Compilation error !
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Invalid escape sequence (valid ones are  \b  \t  \n  \f  \r  \"  \'  \\
)

        at regex.Replacement.main(Replacement.java:15)
```

The following example (splitting by single character):

```
String str = "My1Daddy2cooks34pudding";
String[] words = str.split ("d");
for (String word : words) {
        System.out.println (word);
}
```

gives the following output:

```
My1Da

y2cooks34pu

ing
```

The same string, but with escaped "`d`" (regexp):

```
String str = "My1Daddy2cooks34pudding";
String[] words = str.split ("\\d"); // NOT "d"
for (String word : words) {
        System.out.println (word);
}
```

The output:

```
My
Daddy
cooks

pudding

public String[] split(String regex)
```

Splits this string around matches of the given regular expression. This method works as if by invoking the two-argument `split(...)` method with the given expression and a limit argument of zero. Trailing empty strings are therefore not included in the resulting array. The string "`boo:and:foo`", for example, yields the following results with these expressions:

```
String str = "boo:and:foo";
System.out.println(Arrays.toString(str.split(":")));
System.out.println(Arrays.toString(str.split("o")));
```

The output:

```
[boo, and, foo]
[b, , :and:f]

public String[] split(String regex, int limit)
```

Splits this string around matches of the given regular expression.

The array returned by this method contains each substring of this string that is terminated by another substring that matches the given expression or is terminated by the end of the string. The substrings in the array are in the order in which they occur in this string. If the expression does not match any part of the input then the resulting array has just one element, namely this string.

The `limit` parameter controls the number of times the pattern is applied and therefore affects the length of the resulting array. If the limit `n` is greater than zero then the pattern will be applied at most `n - 1` times, the array's length will be no greater than `n`, and the array's last entry will contain all input beyond the last matched delimiter. If `n` is non-positive then the pattern will be applied as many times as possible and the array can have any length. If `n` is zero then the pattern will be applied as many times as possible, the array can have any length, and trailing empty strings will be discarded:

```
String str = "boo:and:foo";
               System.out.println("a. " + Arrays.toString(str.split(":", 2)));
               System.out.println("b. " + Arrays.toString(str.split(":", 5)));
               System.out.println("c. " + Arrays.toString(str.split(":", -2)));
               System.out.println("d. " + Arrays.toString(str.split("o", 5)));
               System.out.println("e. " + Arrays.toString(str.split("o", -2)));
               System.out.println("f. " + Arrays.toString(str.split("o", 0)));
```

An invocation of this method of the form `str.split(regex, n)` yields the same result as the expression:

```
Pattern.compile(regex).split(str, n)
```

Formatted input

The scanner API provides basic input functionality for reading data from the system console or any data stream. The following example reads a `String` from standard input and expects a following `int` value:

```
Scanner s= new Scanner(System.in);
String param= s.next();
int value=s.nextInt();
s.close();
```

The `Scanner` methods like `next` and `nextInt` will block if no data is available. If you need to process more complex input, then there are also pattern-matching algorithms, available from the `java.util.Formatter` class.

`java.util.Scanner` is 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.

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 allows `long` types to be assigned from entries in a file `myNumbers`:

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

```
}
```

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

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).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 same output can be generated with this code, which uses a regular expression to parse all four tokens at once:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input);
s.findInLine("(\\d+) fish (\\d+) fish (\\w+) fish (\\w+)");
MatchResult result = s.match();
for (int i=1; i<=result.groupCount(); i++) {
        System.out.println(result.group(i));
}
s.close();
```

Class `java.util.Scanner` implements a simple text scanner (lexical analyzer) which uses regular expressions to parse primitive types and strings from its source.

A `Scanner` converts the input from its source into tokens using a delimiter pattern, which by default matches whitespace.

The tokens can be converted into values of different types using the various `next()` methods:

```
Scanner scanner = new Scanner(System.in);     // Connected to standard input.
int i = scanner.nextInt();

Scanner scanner = new Scanner(new File("myLongNumbers"));
                                              // (1) Construct a scanner.
while (scanner.hasNextLong()) {               // (2) End of input? May block.
        long aLong = scanner.nextLong();      // (3) Deal with the current
                                              //     token. May block.
}
scanner.close();                              // (4) Closes the scanner.
                                              //     May close the source.
```

Before parsing the next token with a particular `next()` method, for example at (3), a lookahead can be performed by the corresponding `hasNext()` method as shown at (2).

The `next()` and `hasNext()` methods and their primitive-type companion methods (such as `nextInt()` and `hasNextInt()`) first skip any input that matches the delimiter pattern, and then attempt to return the next token.

Constructing a `Scanner`

A scanner must be constructed to parse text:

```
Scanner(Type source)
```

Returns an appropriate scanner. `Type` can be a `String`, a `File`, an `InputStream`, a `ReadableByteChannel`, or a `Readable` (implemented by `CharBuffer` and various `Readers`).

Scanning

A scanner throws an `InputMismatchException` when the next token cannot be translated into a valid value of requested type.

Lookahead methods:

```
// returns true if this scanner has another token in its input
boolean hasNext()

// returns true if the next token matches the specified pattern
boolean hasNext(Pattern pattern)

// returns true if the next token matches the pattern constructed
// from the specified string
boolean hasNext(String pattern)

// returns true if the next token in this scanner's input can be interpreted as
an
// numeric type value corresponding to 'XXX' in the default or specified
// radix
boolean hasNextXXX()
boolean hasNextXXX(int radix)

// returns true if the next token in this scanner's
// input can be interpreted as a boolean value using
// a case insensitive pattern created from the string
// "true|false"
boolean hasNextBoolean()
```

The name `XXX` can be: `Byte`, `Short`, `Int`, `Long`, `Float`, `Double` or `BigInteger`.

Parsing the next token methods:

```
// scans and returns the next complete token from this scanner
String next()

// returns the next string in the input that matches the specified pattern
String next(Pattern pattern)

// returns the next token if it matches the pattern constructed from the
specified string
String next(String pattern)

// scans the next token of the input as a 'xxx' value corresponding to 'XXX'
xxx nextXXX()
```

```
xxx nextXXX(int radix)

// scans the next token of the input into a boolean
// value and returns that value
boolean nextBoolean()

// advances this scanner past the current line and
// returns the input that was skipped
String nextLine()
```

The name `XXX` can be: `Byte`, `Short`, `Int`, `Long`, `Float`, `Double` or `BigInteger`. The corresponding 'xxx' can be: `byte`, `short`, `int`, `long`, `float`, `double` or `BigInteger`.

Example:

```
String input = "123 45,56 TRUE 567 722 blabla";
Scanner scanner = new Scanner(input);
out.println(scanner.hasNextInt());
out.println(scanner.nextInt());
out.println(scanner.hasNextDouble());
out.println(scanner.nextDouble());
out.println(scanner.hasNextBoolean());
out.println(scanner.nextBoolean());
out.println(scanner.hasNextInt());
out.println(scanner.nextInt());
out.println(scanner.hasNextLong());
out.println(scanner.nextLong());
out.println(scanner.hasNext());
out.println(scanner.next());
out.println(scanner.hasNext());
scanner.close();
```

The output:

```
true
123
true
45.56
true
true
true
567
true
722
true
blabla
false
```

Error in parsing:

```
String input = "123,123";
Scanner scanner = new Scanner(input);
out.println(scanner.hasNextInt());
out.println(scanner.nextInt());
scanner.close();
```

The output (runtime exception):

```
false
```

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        ...
```

Formatted output

Developers now have the option of using printf-type functionality to generate formatted output. This will help migrate legacy C applications, as the same text layout can be preserved with little or no change.

Most of the common C `printf` formatters are available, and in addition some Java classes like `Date` and `BigInteger` also have formatting rules. See `the java.util.Formatter` class for more information. Although the standard UNIX newline '\n' character is accepted, for cross-platform support of newlines the Java %n is recommended. Furthermore, J2SE 5.0 added a `printf()` method to the `PrintStream` class. So now you can use `System.out.printf()` to send formatted numerical output to the console. It uses a `java.util.Formatter` object internally:

```
System.out.printf("name count%n");
System.out.printf("%s %5d%n", user,total);
```

The simplest of the overloaded versions of the method goes as

```
printf (String format, Object... args)
```

The `format` argument is a string in which you embed specifier substrings that indicate how the arguments appear in the output. For example:

```
double pi = Math.PI;
System.out.printf ("1. pi = %5.3f %n", pi);
System.out.printf ("2. pi = %f    %n", pi);
System.out.printf ("3. pi = %b    %n", pi);
System.out.printf ("4. pi = %s    %n", pi);
```

results in the console output:

```
1. pi = 3,142
2. pi = 3,141593
3. pi = true
4. pi = 3.141592653589793
```

The format string includes the specifier "`%5.3f`" that is applied to the argument. The '%' sign signals a specifier. The width value 5 requires at least five characters for the number, the precision value 3 requires three places in the fraction, and the conversion symbol 'f' indicates a decimal representation of a floating-point number.

A specifier needs at least the conversion character, of which there are several besides 'f'. Some of the other conversions include:

- `%b` - If the argument arg is `null`, then the result is "`false`". If arg is a `boolean` or `Boolean`, then the result is the string returned by `String.valueOf()`. Otherwise, the result is "`true`".
- `%c` - The result is a Unicode character.
- `%d` - The result is formatted as a decimal integer.
- `%f` - The result is formatted as a decimal number.
- `%s` - If the argument arg is `null`, then the result is "`null`". If arg implements `Formattable`, then `arg.formatTo` is invoked. Otherwise, the result is obtained by invoking `arg.toString()`.

There are also special conversions for dates and times. The general form of the specifier includes several optional terms:

```
%[argument_index$][flags][width][.precision]conversion
```

The `argument_index` indicates to which argument the specifier applies. For example, `%2$` indicates the second argument in the list. A flag indicates an option for the format. For example, '+' requires that a sign be included and '0' requires padding with zeros. The width indicates the minimum number of characters and the precision is the number of places for the fraction.

There is also one specifier that doesn't correspond to an argument. It is "%n" which outputs a line break. A "`\n`" can also be used in some cases, but since "`%n`" always outputs the correct platform-specific line separator, it is portable across platforms whereas "`\n`" is not.

Classes `java.lang.String`, `java.io.PrintStream`, `java.io.PrintWriter` and `java.util.Formatter` provide the following overloaded methods for formatted output:

```
// Writes a formatted string using the specified format string and argument list
format(String format, Object... args)
format(Locale l, String format, Object... args)
```

The `format()` method returns a `String`, a `PrintStream`, a `PrintWriter` or a `Formatter` respectively for these classes, allowing method call chaining.

The `format()` method is `static` in the `String` class.

In addition, classes `PrintStream` and `PrintWriter` provide the following convenience methods:

```
// Writes a formatted string using the specified format string and argument list
printf(String format, Object... args)
printf(Locale l, String format, Object... args)
```

Format string syntax provides support for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output.

Class `java.util.Formatter` provides the core support for formatted output.

Format string syntax

The format string can specify fixed text and embedded format specifiers:

```
out.printf("Formatted output|%6d|%8.3f|%.2f%n",      2005, Math.PI, 1234.0354);
```

The output will be (default locale Russian):

```
Formatted output|  2005|   3,142|1234,04
```

The format string is the first argument.

It contains three format specifiers `%6d`, `%8.3f`, and `%.2f` which indicate how the arguments should be processed and where the arguments should be inserted in the format string.

All other text in the format string is fixed, including any other spaces or punctuation.

The argument list consists of all arguments passed to the method after the format string. In the above example, the argument list is of size three.

In the above example, the first argument is formatted according to the first format specifier, the second argument is formatted according to the second format specifier, and so on.

Format specifiers for general, character, and numeric types

```
%[argument_index$][flags][width][.precision]conversion
```

The characters `%`, `$` and `.` have special meaning in the context of the format specifier.

The optional `argument_index` is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by "`1$`", the second by "`2$`", and so on.

The optional `flags` is a set of characters that modify the output format. The set of valid flags depends on the conversion.

The optional `width` is a decimal integer indicating the minimum number of characters to be written to the output.

The optional `precision` is a decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion.

The required `conversion` is a character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type.

Conversion categories (required for exam only)

General ('`b`', '`s`'): May be applied to any argument type.

Character ('`c`'): May be applied to basic types which represent unicode characters: `char`, `Character`, `byte`, `Byte`, `short`, and `Short`.

Numeric integral ('`d`'): May be applied to integral types: `byte`, `Byte`, `short`, `Short`, `int`, `Integer`, `long`, `Long`, and `BigInteger`.

```
out.printf("%8d", 10.10); // WRONG !!!

java.util.IllegalFormatConversionException: d != java.lang.Double

out.printf("%8d", (int) 10.10); // OK !
```

Numeric floating point ('`f`'): May be applied to floating-point types: `float`, `Float`, `double`, `Double`, and `BigDecimal`.

```
out.printf("%8.3f", 10); // WRONG !!!

java.util.IllegalFormatConversionException: f != java.lang.Integer
```

Conversion rules

'`b`' - If the argument `arg` is `null`, then the result is "`false`". If `arg` is a `boolean` or `Boolean`, then the result is string returned by `String.valueOf()`. Otherwise, the result is "`true`".

'`c`' - The result is a Unicode character.

'`d`' - The result is formatted as a decimal integer.

'f' - The result is formatted as a floating point decimal number.

's' - If the argument `arg` is `null`, then the result is "`null`". If `arg` implements `Formattable`, then `arg.formatTo()` is invoked. Otherwise, the result is obtained by invoking `arg.toString()`.

Precision

For general argument types, the precision is the maximum number of characters to be written to the output:

```
out.printf("|%8.3s|", "2005");

|     200|
```

For the floating-point conversions: if the conversion is 'f', then the precision is the number of digits after the decimal separator:

```
out.printf("|%8.3f| %n", 2005.1234);
out.printf("|%8.3s|", true);

|2005,123|
|     tru|
```

For character ('c') and integral ('d') argument types the precision is not applicable. If a precision is provided, an exception will be thrown:

```
out.printf("|%8.3d|", 2005);

Exception in thread "main" java.util.IllegalFormatPrecisionException: 3
```