C. `test case live2`
D. Compilation fails.
E. An exception is thrown at runtime.
Answer:
- **E** is correct. **Because the short circuit `(||)` is not used, both operands are evaluated. Since `args[1]` is past the args array bounds, an `ArrayIndexOutOfBoundsException` is thrown.**
- **A, B, C,** and **D** are incorrect based on the above. (Objective 7.6)

- The ! operator (called the "inversion" operator), returns the opposite value of the `boolean` operand it precedes.

Given:
```
1. class Maybe {
2.     public static void main(String[] args) {
3.           boolean b1 = true;
4.           boolean b2 = false;
5.           System.out.print(!false ^ false);
6.           System.out.print(" " + (!b1 & (b2 = true)));
7.           System.out.println(" " + (b2 ^ b1));
8.     }
9. }
```
Which are true?
A. Line 5 produces `true`.
B. Line 5 produces `false`.
C. Line 6 produces `true`.
D. Line 6 produces `false`.
E. Line 7 produces `true`.
F. Line 7 produces `false`.
Answer:
- **A , D,** and **F** is correct. The `^ (xor)` returns true if exactly one operand is `true`. The `!` inverts the operand's `boolean` value. On line 6 `b2 = true` is an assignment not a comparison, and it's evaluated because `&` does not short-circuit it.
- **B, C,** and **E** are incorrect based on the above. (Objective 7.6)

Flow control, Exceptions & Assertion

Writing Code Using if and switch Statements (Obj. 2.1)

- The only legal expression in an `if` statement is a `boolean` expression, in other words an expression that resolves to a `boolean` or a `boolean` variable.
- **Watch out for `boolean` assignments (=) that can be mistaken for `boolean` equality (==) tests:**
```
boolean x = false;
if (x = true) { } // an assignment, so x will always be true!
```
- Curly braces are optional for `if` blocks that have only one conditional statement. But watch out for misleading indentations.
- **`switch` statements can evaluate only to `enum`s or the `byte`, `short`, `int`, and `char` data types. You can't say,**
```
long s = 30;
switch(s) { } // compiler error!
```
- **The `case` constant must be a literal or `final` variable, or a constant expression, including an `enum`. You cannot have a case that includes a nonfinal variable, or a range of values.**

Example1: A `case` constant must evaluate to the same type as the `switch` expression can use, with one additional—and big—constraint: the `case` constant must be a compile time constant!

```
final int a = 1; // a compile time constant
final int b;
b = 2; // a final variable can be initialized at runtime
int x = 0;
switch (x) {
       case a: // ok
       case b: // compiler error
}
```

Example2 :

```
byte g = 2;
switch(g) {
       case 23:
       case 128: // compiler error! 128 too large for byte
}
```

Example3: It's also illegal to have more than one `case` label using the same value.

```
int temp = 90;
switch(temp) {
       case 80 : System.out.println("80");
       case 80 : System.out.println("80"); // won't compile!
       case 90 : System.out.println("90");
       default : System.out.println("default");
}
```

Example4: Use of Enums.

```
class Swill {
       public static void main(String[] args) {
              String s = "-";
              switch(TimeZone.CST) {
                     case EST: s += "e";
                     case CST: s += "c";
                     case MST: s += "m";
                     default: s += "X";
                     case PST: s += "p";
              }
              System.out.println(s);
       }
}
enum TimeZone {EST, CST, MST, PST }
```

O/P is: -cmXp

- If the condition in a `switch` statement matches a `case` constant, execution will run through all code in the `switch` following the matching `case` statement until a `break` statement or the end of the `switch` statement is encountered. In other words, the matching `case` is just the entry point into the `case` block, but unless there's a `break` statement, the matching `case` is not the only `case` code that runs. – **Fall-through**.
- The `default` keyword should be used in a `switch` statement if you want to run some code when none of the `case` values match the conditional value.
- **The `default` block can be located anywhere in the `switch` block**, so if no `case` matches, the `default` block will be entered, and if the `default` does not contain a `break`, then code will continue to execute (fall-through) to the end of the `switch` or until the `break` statement is encountered.

```
int x = 2;
switch (x) {
       case 2: System.out.println("2");
       default: System.out.println("default");
       case 3: System.out.println("3");
```

```
        case 4: System.out.println("4");
}
```

Given:

```
1. class Ring {
2.     final static int x2 = 7;
3.     final static Integer x4 = 8;
4.     public static void main(String[] args) {
5.           Integer x1 = 5;
6.           String s = "a";
7.           if(x1 < 9) s += "b";
8.           switch(x1) {
9.                 case 5: s += "c";
10.                case x2: s += "d";
11.                case x4: s += "e";
12.          }
13.          System.out.println(s);
14.    }
15. }
```

What is the result?

A. `abc`
B. `abcde`
C. Compilation fails due only to an error on line 7.
D. Compilation fails due only to an error on line 8.
E. Compilation fails due only to an error on line 10.
F. Compilation fails due only to an error on line 11.
G. Compilation fails due to errors on multiple lines.

Answer:

- **F** is correct. A `switch` statement requires its `case` expressions to be constants, and wrapper variables (even `final static` ones) aren't considered constants. The rest of the code is correct.
- **A, B, C, D, E,** and **G** are incorrect based on the above. (Objective 2.1)

Writing Code Using Loops (Objective 2.2)

- A basic `for` statement has three parts: declaration and/or initialization, boolean evaluation, and the iteration expression.

```
for (int x = 0; (x > 5), (y < 2); x++) { } // Compiler Error: too many
                                           // expressions
```

- If a variable is incremented or evaluated within a basic `for` loop, it must be declared before the loop, or within the `for` loop declaration.
- A variable declared (not just initialized) within the basic `for` loop declaration cannot be accessed outside the `for` loop (in other words, code below the `for` loop won't be able to use the variable).

```
for (int x = 3; x < 20; x++) { }
System.out.println(x); // Compiler Error: Cannot resolve symbol x
```

- You can initialize more than one variable of the same type in the first part of the basic `for` loop declaration; each initialization must be separated by a comma.

```
for (int x = 10, y = 3; y > 3; y++) { }
```

- An enhanced `for` statement (new as of Java 5), has two parts, the *declaration* and the *expression*. It is used only to loop through arrays or collections.
- With an enhanced `for`, the *expression* is the array or collection through which you want to loop.

```
int [] a = {1,2,3,4};
for(int n : a) // enhanced for loop
System.out.print(n);
```

- With an enhanced `for`, the *declaration* is the block variable, whose type is compatible with the elements of the array or collection, and that variable contains the value of the element for the given iteration.

```
int x;
long x2;
int[] arr = {7,6,5,4,3,2,1};
Long [] La = {4L, 5L, 6L};
long [] la = {7L, 8L, 9L};
int [][] twoDee = {{1,2,3}, {4,5,6}, {7,8,9}};
String [] sNums = {"one", "two", "three"};
Animal [] animals = {new Dog(), new Cat()};

// Legal 'for' declarations
for(long y : la ) ; // loop thru an array of longs
for(long lp : La) ; // autoboxing the Long objects into longs
for(int[] n : twoDee) ; // loop thru the array of arrays
for(int n2 : twoDee[2]) ; // loop thru the 3rd sub-array
for(String s : sNums) ; // loop thru the array of Strings
for(Object o : sNums) ; // set an Object reference to
// each String
for(Animal a : animals) ; // set an Animal reference to each element

// ILLEGAL 'for' declarations
for(x2 : la) ; // x2 is already declared
for(int x2 : twoDee) ; // can't stuff an array into an int
for(int x3 : la) ; // can't stuff a long into an int
for(Dog d : animals) ; // you might get a Cat!
int y = 0; for(y: arr) {} // enhanced for must declare its first operand.
```

- You cannot use a number (old C-style language construct) or anything that does not evaluate to a `boolean` value as a condition for an `if` statement or looping construct. You can't, for example, say `if(x)`, unless `x` is a `boolean` variable.
- The `do` loop will enter the body of the loop at least once, even if the test condition is not met.

Using break and continue (Objective 2.2)

- An unlabeled `break` statement will cause the current iteration of the innermost looping construct to stop and the line of code following the loop to run.
- An unlabeled `continue` statement will cause: the current iteration of the innermost loop to stop, the condition of that loop to be checked, and if the condition is met, the loop to run again.
- If the `break` statement or the `continue` statement is labeled, it will cause similar action to occur on the labeled loop, not the innermost loop.

Handling Exceptions (Objectives 2.4, 2.5, and 2.6)

- Exceptions come in two flavors: **checked** and **unchecked**.
- Checked exceptions include all subtypes of `Exception`, excluding classes that extend `RuntimeException`.
- **Checked exceptions are subject to the handle or declare rule;** any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using `throws`, or handle the exception with an appropriate try/catch.
- **Subtypes of `Error` or `RuntimeException` are unchecked,** so the **compiler doesn't enforce the handle or declare rule. You're free to handle them, or to declare them, but the compiler doesn't care one way or the other.**

- If you use an optional `finally` block, it will always be invoked, regardless of whether an exception in the corresponding `try` is thrown or not, and regardless of whether a thrown exception is caught or not.
- The only exception to the `finally`-will-always-be-called rule is **that a `finally` will not be invoked if the JVM shuts down. That could happen if code from the `try` or `catch` blocks calls `System.exit()`.**

```
try {
      // do stuff
} catch (SomeException ex) {
      // do exception handling
} finally {
      // clean up
}
```

- Just because `finally` is invoked does not mean it will complete. **Code in the `finally` block could itself raise an exception or issue a `System.exit()`.**
- Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to `main()`, and `main()` is "ducking" the exception by declaring it).
- You can create your own exceptions, normally by extending `Exception` or one of its subtypes. Your exception will then be considered a checked exception, and the compiler will enforce the handle or declare rule for that exception.
- **All `catch` blocks must be ordered from most specific to most general.** If you have a `catch` clause for both `IOException` and `Exception`, you must put the `catch` for `IOException` first in your code. Otherwise, the `IOException` would be caught by `catch(Exception e)`, because a `catch` argument can catch the specified exception or any of its subtypes! **The compiler will stop you from defining `catch` clauses that can never be reached.**

Given:
```
try { int x = Integer.parseInt("two"); }
```
Which could be used to create an appropriate `catch` block? (Choose all that apply.)
A. `ClassCastException`
B. `IllegalStateException`
C. `NumberFormatException`
D. `IllegalArgumentException`
E. `ExceptionInInitializerError`
F. `ArrayIndexOutOfBoundsException`
Answer:
- **C** and **D** are correct. `Integer.parseInt` can throw a `NumberFormatException`, and `IllegalArgumentException` is its superclass (i.e., a broader exception).
- **A, B, E,** and **F** are not in `NumberFormatException`'s class hierarchy. (Objective 2.6)

Given:
```
1. class Ping extends Utils {
2.     public static void main(String [] args) {
3.           Utils u = new Ping();
4.           System.out.print(u.getInt(args[0]));
5.     }
6.     int getInt(String arg) {
7.           return Integer.parseInt(arg);
8.     }
9. }
10. class Utils {
11.    int getInt(String x) throws Exception { return 7; }
```

```
12. }
```

And the following three possible changes:
C1. Declare that `main()` throws an `Exception`.
C2. Declare that `Ping.getInt()` throws an `Exception`.
C3. Wrap the invocation of `getInt()` in a `try / catch` block.
Which change(s) allow the code to compile? (Choose all that apply.)
A. Just C1 is sufficient.
B. Just C2 is sufficient.
C. Just C3 is sufficient.
D. Both C1 and C2 are required.
E. Both C1 and C3 are required.
F. Both C2 and C3 are required.
G. All three changes are required.
Answer:

- **A** and **C** are correct. Remember that line 4 is making a polymorphic call so the compiler knows that an exception might be thrown. If C1 is implemented the exception has been sufficiently declared, and if C3 is implemented the exception has been sufficiently handled. C2 is not necessary in either case. NumberFormatException is an unchecked exception so even though the parseInt() can throw the NumberFormatException, the compiler does not force us to handle or throw that exception.
- **B, D, E, F,** and **G** are incorrect based on the above. (Objective 2.4)

- Some exceptions are created by programmers, some by the JVM.

TABLE 5-2 Descriptions and Sources of Common Exceptions.

Exception (Chapter Location)	Description	Typically Thrown
ArrayIndexOutOfBoundsException (Chapter 3, "Assignments")	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array).	By the JVM
ClassCastException (Chapter 2, "Object Orientation")	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.	By the JVM
IllegalArgumentException (Chapter 3, "Assignments")	Thrown when a method receives an argument formatted differently than the method expects.	Programmatically
IllegalStateException (Chapter 6, "Formatting")	Thrown when the state of the environment doesn't match the operation being attempted, e.g., using a Scanner that's been closed.	Programmatically
NullPointerException (Chapter 3, "Assignments")	Thrown when attempting to access an object with a reference variable whose current value is `null`.	By the JVM
NumberFormatException (Chapter 6, "Formatting")	Thrown when a method that converts a String to a number receives a String that it cannot convert.	Programmatically
AssertionError (This chapter)	Thrown when a statement's boolean test returns `false`.	Programmatically
ExceptionInInitializerError (Chapter 3, "Assignments")	Thrown when attempting to initialize a static variable or an initialization block.	By the JVM
StackOverflowError (This chapter)	Typically thrown when a method recurses too deeply. (Each invocation is added to the stack.)	By the JVM
NoClassDefFoundError (Chapter 10, "Development")	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing `.class` file.	By the JVM

Which are most typically thrown by an API developer or an application developer as opposed to being thrown by the JVM? (Choose all that apply.)

A. `ClassCastException`
B. `IllegalStateException`
C. `NumberFormatException`
D. `IllegalArgumentException`
E. `ExceptionInInitializerError`

Answer:

- **B , C,** and **D** are correct. **B** is typically used to report an environment problem such as trying to access a resource that's closed. **C** is often thrown in API methods that attempt to convert poorly formed String arguments to numeric values. **D** is often thrown in API methods that receive poorly formed arguments.
- **A** and **E** are thrown by the JVM. (Objective 2.6)

Working with the Assertion Mechanism (Objective 2.3)

- Assertions give you a way to test your assumptions during development and debugging.
  Really simple:

```
private void doStuff() {
        assert (y > x);
        // more code assuming y is greater than x
}
```

  Simple:

```
private void doStuff() {
        assert (y > x): "y is " + y + " x is " + x;
        // more code assuming y is greater than x
}
```

- **Assertions are typically enabled during testing but disabled during deployment.**
- You can use `assert` as a keyword (as of version 1.4) or an identifier, but not both together. To compile older code that uses `assert` as an identifier (for example, a method name), use the `-source 1.3` command-line flag to `javac`.
- **Assertions are disabled at runtime by default. To enable them, use a command- line flag `-ea` or `-enableassertions`.**
- Selectively **disable assertions by using the `-da` or `-disableassertions` flag**.
- If you enable or disable assertions using the flag without any arguments, you're enabling or disabling assertions in general. You can combine enabling and disabling switches to have assertions enabled for some classes and/or packages, but not others.
- You can enable and disable assertions on a class-by-class basis, using the following syntax:
  - **`java -ea -da:MyClass TestClass`**
- You can enable and disable assertions on a package-by-package basis, and any package you specify also includes any subpackages (packages further down the directory hierarchy).

TABLE 5-4 Assertion Command-Line Switches

Command-Line Example	What It Means
`java -ea` `java -enableassertions`	Enable assertions.
`java -da` `java -disableassertions`	Disable assertions (the default behavior of 1.5).
`java -ea:com.foo.Bar`	Enable assertions in class `com.foo.Bar`.
`java -ea:com.foo...`	Enable assertions in package `com.foo` and any of its subpackages.
`java -ea -dsa`	Enable assertions in general, but disable assertions in system classes.
`java -ea -da:com.foo...`	Enable assertions in general, but disable assertions in package `com.foo` and any of its subpackages.

- **Do not use assertions to validate arguments to `public` methods.**

```
public void doStuff(int x) {
        assert (x > 0); // inappropriate !
        // do things with x
}
```

  You don't want publicly accessible code that works only conditionally, depending on whether assertions are enabled. If you need to validate `public` method arguments, you'll probably use exceptions to throw, say, an `IllegalArgumentException` if the values passed to the `public` method are invalid.

- **Do not use `assert` expressions that cause side effects.** Assertions aren't guaranteed to always run, and you don't want behavior that changes depending on whether assertions are enabled.

```
public void doStuff() {
       assert (modifyThings());
       // continues on
}
public boolean modifyThings() {
       y = x++;
       return true;
}
```

The rule is, an `assert` expression should leave the program in the same state it was in before the expression!

- **Do use assertions—even in `public` methods—to validate that a particular code block will never be reached. You can use `assert false;` for code that should never be reached, so that an assertion error is thrown immediately if the `assert` statement is executed.**

```
switch(x) {
       case 1: y = 3;
       case 2: y = 9;
       case 3: y = 27;
       default: assert false; // We're never supposed to get here!
}
```

Now some legal and illegal assertions:

```
void noReturn() { }
int aReturn() { return 1; }
void go() {
int x = 1;
boolean b = true;

// the following six are legal assert statements
assert(x == 1);
assert(b);
assert true;
assert(x == 1) : x;
assert(x == 1) : aReturn();
assert(x == 1) : new ValidAssert();

// the following six are ILLEGAL assert statements
assert(x = 1); // none of these are booleans
assert(x);
assert 0;
assert(x == 1) : ; // none of these return a value
assert(x == 1) : noReturn();
assert(x == 1) : ValidAssert va;
}
```

Given two files:

```
1. class One {
2. public static void main(String[] args) {
3. int assert = 0;
4. }
5. }
```

```
1. class Two {
2. public static void main(String[] args) {
3. assert(false);
4. }
5. }
```

And the four command-line invocations:

```
javac -source 1.3 One.java
```

```
javac -source 1.4 One.java
javac -source 1.3 Two.java
javac -source 1.4 Two.java
```

What is the result? (Choose all that apply.)

A. Only one compilation will succeed.
B. Exactly two compilations will succeed.
C. Exactly three compilations will succeed.
D. All four compilations will succeed.
E. No compiler warnings will be produced.
F. At least one compiler warning will be produced.
Answer:

- **B** and **F** are correct. Class One will compile (and issue a warning) using the `1.3` flag, and class Two will compile using the `1.4` flag.
- **A, C, D,** and **E** are incorrect based on the above. (Objective 2.3)

Strings, I/O, Formatting & Parsing

Using String, StringBuffer, and StringBuilder (Objective 3.1)

- String objects are immutable, and String reference variables are not.
- If you create a new String without assigning it, it will be lost to your program.
- If you redirect a String reference to a new String, the old String can be lost.
- String methods use zero-based indexes, except for the second argument of `substring()`.
- **The String class is `final`**—its methods can't be overridden.
- When the JVM finds a String literal, it is added to the **String literal pool**. To make Java more memory efficient, the JVM sets aside a special area of memory called the "String constant pool." When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created. (The existing String simply has an additional reference.) Now we can start to see why making String objects **immutable** is such a good idea. If several reference variables refer to the same String without even knowing it, it would be very bad if any of them could change the String's value. String literal pool is also a reason why String class is **final** (and hence none of its funcationality can be overridden).

```
String s = "abc"; // creates one String object and one reference variable
```

In this simple case, "abc" will go in the pool and `s` will refer to it.

```
String s = new String("abc"); // creates two objects, and one reference
                              // variable.
```

In this case, because we used the `new` keyword, Java will create a new String object in normal (non-pool) memory, and `s` will refer to it. In addition, the literal "`abc`" will be placed in the pool.

- Strings have a method: `length()`, arrays have an attribute named `length`.
- **The StringBuffer's API is the same as the new StringBuilder's API, except that StringBuilder's methods are not synchronized for thread safety.**

A common use for StringBuffers and StringBuilders is file I/O when large, ever-changing streams of input are being handled by the program. In these cases, large blocks of characters are handled as units, and StringBuffer objects are the ideal way to handle a

block of data, pass it on, and then reuse the same memory to handle the next block of data.

- StringBuilder methods should run faster than StringBuffer methods.
- All of the following bullets apply to both StringBuffer and StringBuilder:
  - They are **mutable**—they can change without creating a new object.
  - StringBuffer methods act on the invoking object, and objects can change without an explicit assignment in the statement.
  - **StringBuffer `equals()` is not overridden**; it doesn't compare values.
- Remember that chained methods are evaluated from left to right.

```
String y = x.concat("def").toUpperCase().replace('C','x');
//chained methods
System.out.println("y = " + y); // result is "y = ABxDEF"
```

- **String methods** to remember:
  - `charAt()`,
  - `concat()`, -> `s = s.concat(" more stuff");`
  - `equalsIgnoreCase()`,
  - `length()`,
  - `replace()`,
  - `substring()`,
  - `toLowerCase()`,
  - `toString()`,
  - `toUpperCase()`, and
  - `trim()`.
- **Stringbuffer methods** to remember:
  - `append()`,
  - `delete()`,
  - `insert()`,
  - `reverse()`, and
  - `toString()`.

File I/O (Objective 3.2)

- The classes you need to understand in java.io are
  - File,
  - FileReader,
  - BufferedReader,
  - FileWriter,
  - BufferedWriter, and
  - PrintWriter.

Byte Stream classes (FileInputStream/FileOutputStream) are used to read and write bytes, and Readers and Writers are used to read and write characters. I/O objective for exam only requires character stream classes.

- A new File object doesn't mean there's a new file on your hard drive.
- File objects can represent either a file or a directory.
- **The File class lets you manage (add, rename, and delete) files and directories.**
- The methods **`createNewFile()`** and **`mkDir()`** add entries to your file system.
- FileWriter and FileReader are low-level I/O classes. You can use them to write and read files, but they should usually be wrapped.
- Classes in java.io are designed to be "chained" or "wrapped." (This is a common use of the **decorator** design pattern.)

- It's very common to "wrap" a **BufferedReader around a FileReader,** to get access to higher-level (more convenient) methods like **readLine**().
- **PrintWriters can be used to wrap other Writers, but as of Java 5 they can be built directly from Files or Strings.**
- Java 5 **PrintWriters** have new **`append()`, `format()`, and `printf()`** methods.

TABLE 6-1 java.io Mini API

java.io Class	Extends From	Key Constructor(s) Arguments	Key Methods
File	Object	File, String String String, String	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
BufferedWriter	Writer	Writer	close() flush() newLine() write()
PrintWriter	Writer	File (as of Java 5) String (as of Java 5) OutputStream Writer	close() flush() format()*, printf()* print(), println() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()
			*Discussed later

Usage:

```
import java.io.*;
import static java.lang.System.*;

public class FileIO
{

        public static void main(String[] args)
        {
                PrintWriter pw = null;
                BufferedReader br = null;

                try {
                        // create a File object
                        File myDir = new File("testDir");
                        if (!myDir.exists()) {myDir.mkdir();}
                        File file = new File(myDir, "fileWrite2.txt");
                        if (!file.exists()) {
                                if(file.createNewFile()) {
                                out.println("New file created - " + file.getName());
                                }
                        }
                        // create a FileWriter that will send its output to a File
                        //FileWriter fw = new FileWriter(file);

                        // Alternatively, pass fw after uncommenting line above
                        pw = new PrintWriter(file);

                        pw.println("howdy"); // write the data
                        pw.println("folks");
```

```
                        pw.flush();

                        // create a FileReader to get data from 'file'
                        FileReader fr = new FileReader(file);
                        // create a BufferReader to get its data from a Reader
                        br = new BufferedReader(fr);
                        String data = br.readLine(); // read some data

                        while(data != null) {
                                out.println(data);
                                data = br.readLine();
                        }

                        String[] list = myDir.list();
                        for(String s: list)
                                out.println(s);

                        File newDir = new File("newTestDir");
                        myDir.renameTo(newDir);

                        File newFile = new File(newDir, "newFile.txt");
                        file.renameTo(newFile);

                        newFile.delete();
                        newDir.delete(); // if deleting a non-empty directory,
                                         // delete() returns false.

                } catch(FileNotFoundException fnfe) {
                        fnfe.printStackTrace();
                } catch(IOException ioe) {
                        ioe.printStackTrace();
                } finally {
                        if (pw != null) pw.close();
                        try {
                                if (br != null) br.close();
                        } catch(IOException ioe) {ioe.printStackTrace();}
                }
        }
}
```

Serialization (Objective 3.3)

- The classes you need to understand are all in the java.io package; they include: **ObjectOutputStream** and **ObjectInputStream** primarily, and **FileOutputStream** and **FileInputStream** because you will use them to create the low-level streams that the ObjectXxxStream classes will use.
- A class must **implement the Serializable interface** before its objects can be serialized.
- The **`ObjectOutputStream.writeObject()`** method serializes objects, and the **`ObjectInputStream.readObject()`** method deserializes objects.
- If you mark an instance variable **`transient`**, it will not be serialized even thought the rest of the object's state will be.
- You can supplement a class's automatic serialization process by implementing the `writeObject()` and `readObject()` methods. If you do this, embedding calls to `defaultWriteObject()` and `defaultReadObject()`, respectively, will handle the part of serialization that happens normally.
- **If a superclass implements Serializable, then its subclasses do automatically.**
- **IMP: If a superclass doesn't implement Serializable, then when a subclass object is deserialized, the superclass constructor will run.**
- DataInputStream and DataOutputStream aren't actually on the exam, in spite of what the Sun objectives say.

import java.io.*;

public class SuperNotSerial

```
{
    public static void main(String [] args) {
            Collar c = new Collar(20);
            Dog d = new Dog(c, 35, "Fido");
            System.out.println("before: Name = " + d.name +
                            " Weight = " + d.weight +
                            " CollarSize = " +  d.theCollar.getCollarSize());
            try {
                    FileOutputStream fs = new FileOutputStream("testSer.ser");
                    ObjectOutputStream os = new ObjectOutputStream(fs);
                    os.writeObject(d); // this results in call to writeObject implemented in Dog.
                    os.close();
            } catch (Exception e) {
                    e.printStackTrace();
            }
            try {
                    FileInputStream fis = new FileInputStream("testSer.ser");
                    ObjectInputStream ois = new ObjectInputStream(fis);
                    d = (Dog) ois.readObject();
                    ois.close();
            } catch (Exception e) {
                    e.printStackTrace();
            }
            System.out.println("before: Name = " + d.name +
                            " Weight = " + d.weight +
                            " CollarSize = " + d.theCollar.getCollarSize());
    }
}

class Animal // Dog IS-A Animal
{ // not serializable !
    int weight = 42;
}

class Collar // Dog HAS-A Collar instance.
{
    private int collarSize;

    Collar(int size)
    {
            collarSize = size;
    }

    int getCollarSize()
    {
            return collarSize;
    }
}

// Two concepts to remember:
// 1. Derived class is serializable, but base class is not. So at deserialization time,
// the base class's constructor will be called thus resetting the value of
// inherited weight value for Dog to 42.
```

```
// If a superclass is Serializable, then according to normal Java interface
// rules, all subclasses of that class automatically implement Serializable implicitly.


// 2. Also we are overriding the writeObject and readObject (note they are private)
// to do partial serialization manually for the Collar instance in dog. For weight
// and name, the defaultWrite/ReadObject methods do the serialization.

// If we did not want to do any serialization manually, then we could ignore
// overriding write/read Object methods.

class Dog extends Animal implements Serializable
{
    String name;
    transient Collar theCollar; // we cannot serialize this.

    Dog(Collar c, int w, String n)
    {
            weight = w; // inherited
            name = n; // not inherited
            theCollar = c;
    }

    // throws IOException
    private void writeObject(ObjectOutputStream out)
    {
            try {
                    out.defaultWriteObject();
                    out.writeInt(theCollar.getCollarSize());
            } catch(Exception e) {
                    e.printStackTrace();
            }
    }

    // throws IOException, ClassNotFoundException
    private void readObject(ObjectInputStream in)
    {
            try {
                    in.defaultReadObject();
                    theCollar = new Collar(in.readInt());
            } catch(Exception e) {
                    e.printStackTrace();
            }
    }
}
```

The output is:
before: Name = Fido Weight = 35 CollarSize = 20
before: Name = Fido Weight = **42** CollarSize = 20

Dates, Numbers, and Currency (Objective 3.4)

- The classes you need to understand are

    - java.util.Date,
    - java.util.Calendar,
    - java.text.DateFormat,
    - java.text.NumberFormat, and
    - java.util.Locale.
- Most of the Date class's methods have been deprecated.
- **A Date is stored as a `long`**, the number of milliseconds since January 1, 1970.
- Date objects are go-betweens the Calendar and Locale classes.
- The **Calendar provides a powerful set of methods to manipulate dates, performing tasks such as getting days of the week, or adding some number of months or years** (or other increments) to a date.
- Create Calendar instances using static factory methods (`getInstance()`).
- The **Calendar methods** you should understand are **`add()`**, which allows you to add or subtract various pieces (minutes, days, years, and so on) of dates, and **`roll()`**, which works like `add()` but doesn't increment a date's bigger pieces. (For example: adding 10 months to an October date changes the month to August, but doesn't increment the Calendar's year value.)
- **DateFormat** instances are created using static factory methods (`getInstance()` and `getDateInstance()`).
- There are several format "styles" available in the DateFormat class.
- DateFormat styles can be applied against various Locales to create a wide array of outputs for any given date.
- The **`DateFormat.format()`** method is used to create Strings containing properly formatted dates.
- The **Locale** class **is used in conjunction with DateFormat and NumberFormat**.
- **Both DateFormat and NumberFormat objects can be constructed with a specific, immutable Locale.**
- For the exam you should understand creating Locales using language, or a combination of language and country.

TABLE 6-2 Common Use Cases When Working with Dates and Numbers

Use Case	Steps
Get the current date and time.	1. Create a Date: `Date d = new Date();` 2. Get its value: `String s = d.toString();`
Get an object that lets you perform date and time calculations in your locale.	1. Create a Calendar: `Calendar c = Calendar.getInstance();` 2. Use `c.add(...)` and `c.roll(...)` to perform date and time manipulations.

Get an object that lets you perform date and time calculations in a different locale.	1. Create a Locale:
	`Locale loc = new Locale(language);` or
	`Locale loc = new Locale(language, country);`
	2. Create a Calendar for that locale:
	`Calendar c = Calendar.getInstance(loc);`
	3. Use `c.add(...)` and `c.roll(...)` to perform date and time manipulations.

Get an object that lets you perform date and time calculations, and then format it for output in different locales with different date styles.	1. Create a Calendar:
	`Calendar c = Calendar.getInstance();`
	2. Create a Locale for each location:
	`Locale loc = new Locale(...);`
	3. Convert your Calendar to a Date:
	`Date d = c.getTime();`
	4. Create a DateFormat for each Locale:
	`DateFormat df = DateFormat.getDateInstance (style, loc);`
	5. Use the format() method to create formatted dates:
	`String s = df.format(d);`

Get an object that lets you format numbers or currencies across many different locales.	1. Create a Locale for each location:
	`Locale loc = new Locale(...);`
	2. Create a NumberFormat:
	`NumberFormat nf = NumberFormat.getInstance(loc);`
	`-or- NumberFormat nf = NumberFormat. getCurrencyInstance(loc);`
	3. Use the format() method to create formatted output:
	`String s = nf.format(someNumber);`

```
import java.util.*;
import java.text.*;
import static java.lang.System.*;

public class DateTest
{
	public static void main(String[] args)
	{
		printCurrentTime();
		doCalendarOperation();
		doGregorianCalendarOperation();
		doDateLocaleOperation();
		doNumberFormatOperation();
	}

	public static void printCurrentTime()
	{
		Date d = new Date();
```

```
            out.println("Current Date and Time: " + d.toString());
    }

    public static void doCalendarOperation()
    {
            Locale loc = new Locale("en","US");
            Calendar cal = Calendar.getInstance(loc);
            out.println("Incrementing date by 30 days...");
            cal.add(Calendar.DATE,30);
            DateFormat df = DateFormat.getDateInstance(DateFormat.FULL, loc);
            out.println(df.format(cal.getTime()));
            out.println("Rolling Date by another 28 days...");
            cal.roll(Calendar.DATE, 28);
            out.println(df.format(cal.getTime()));

            DateFormat df1 = DateFormat.getDateInstance(DateFormat.SHORT);
            String s = df1.format(cal.getTime());
            out.println(s);
            try {
                    Date d2 = df1.parse(s);
                    out.println("parsed = " + d2.toString());
            } catch (ParseException pe) {
                    out.println("parse exc");
            }
    }

    // This method is NOT REQD. for exam!
    public static void doGregorianCalendarOperation()
    {
            // get the supported ids for GMT+05:30 (Indian Standard Time)
            int timeZoneInMillis = (int)(5.5 * 60 * 60 * 1000);
            String[] ids = TimeZone.getAvailableIDs(timeZoneInMillis);
            // if no ids were returned, something is wrong. get out.
            if (ids.length == 0)
               System.exit(0);

             // begin output
            out.println("Current Time");

            // create a Indian Standard Time time zone
            SimpleTimeZone ist = new SimpleTimeZone(timeZoneInMillis, ids[0]);

            // set up rules for daylight savings time
            ist.setStartRule(Calendar.APRIL, 1, Calendar.SUNDAY, 2 * 60 * 60 * 1000);
            ist.setEndRule(Calendar.OCTOBER, -1, Calendar.SUNDAY, 2 * 60 * 60 * 1000);

            // create a GregorianCalendar with the Indian Daylight time zone
            // and the current date and time
            Calendar calendar = new GregorianCalendar(ist);
            Date trialTime = new Date();
            calendar.setTime(trialTime);

            // print out a bunch of interesting things
```

```
                out.println("ERA: " + calendar.get(Calendar.ERA));
                out.println("YEAR: " + calendar.get(Calendar.YEAR));
                out.println("MONTH: " + calendar.get(Calendar.MONTH));
                out.println("WEEK_OF_YEAR: " + calendar.get(Calendar.WEEK_OF_YEAR));
                out.println("WEEK_OF_MONTH: " +
                            calendar.get(Calendar.WEEK_OF_MONTH));
                out.println("DATE: " + calendar.get(Calendar.DATE));
                out.println("DAY_OF_MONTH: " + calendar.get(Calendar.DAY_OF_MONTH));
                out.println("DAY_OF_YEAR: " + calendar.get(Calendar.DAY_OF_YEAR));
                out.println("DAY_OF_WEEK: " + calendar.get(Calendar.DAY_OF_WEEK));
                out.println("DAY_OF_WEEK_IN_MONTH: "
                        + calendar.get(Calendar.DAY_OF_WEEK_IN_MONTH));
                out.println("AM_PM: " + calendar.get(Calendar.AM_PM));
                out.println("HOUR: " + calendar.get(Calendar.HOUR));
                out.println("HOUR_OF_DAY: " + calendar.get(Calendar.HOUR_OF_DAY));
                out.println("MINUTE: " + calendar.get(Calendar.MINUTE));
                out.println("SECOND: " + calendar.get(Calendar.SECOND));
                out.println("MILLISECOND: " + calendar.get(Calendar.MILLISECOND));
                out.println("ZONE_OFFSET: "
                        + (calendar.get(Calendar.ZONE_OFFSET)/(60*60*1000)));
                out.println("DST_OFFSET: "
                        + (calendar.get(Calendar.DST_OFFSET)/(60*60*1000)));

                out.println("Current Time, with hour reset to 3");
                calendar.clear(Calendar.HOUR_OF_DAY); // so doesn't override
                calendar.set(Calendar.HOUR, 3);
                out.println("ERA: " + calendar.get(Calendar.ERA));
                out.println("YEAR: " + calendar.get(Calendar.YEAR));
                out.println("MONTH: " + calendar.get(Calendar.MONTH));
                out.println("WEEK_OF_YEAR: " + calendar.get(Calendar.WEEK_OF_YEAR));
                out.println("WEEK_OF_MONTH: " +
                            calendar.get(Calendar.WEEK_OF_MONTH));
                out.println("DATE: " + calendar.get(Calendar.DATE));
                out.println("DAY_OF_MONTH: " + calendar.get(Calendar.DAY_OF_MONTH));
                out.println("DAY_OF_YEAR: " + calendar.get(Calendar.DAY_OF_YEAR));
                out.println("DAY_OF_WEEK: " + calendar.get(Calendar.DAY_OF_WEEK));
                out.println("DAY_OF_WEEK_IN_MONTH: "
                        + calendar.get(Calendar.DAY_OF_WEEK_IN_MONTH));
                out.println("AM_PM: " + calendar.get(Calendar.AM_PM));
                out.println("HOUR: " + calendar.get(Calendar.HOUR));
                out.println("HOUR_OF_DAY: " + calendar.get(Calendar.HOUR_OF_DAY));
                out.println("MINUTE: " + calendar.get(Calendar.MINUTE));
                out.println("SECOND: " + calendar.get(Calendar.SECOND));
                out.println("MILLISECOND: " + calendar.get(Calendar.MILLISECOND));
                out.println("ZONE_OFFSET: "
                    + (calendar.get(Calendar.ZONE_OFFSET)/(60*60*1000))); // in hours
                out.println("DST_OFFSET: "
                    + (calendar.get(Calendar.DST_OFFSET)/(60*60*1000))); // in hours

        }

        public static void doDateLocaleOperation()
        {
```

```
                Calendar c = Calendar.getInstance();
                c.set(2010, 11, 14); // December 14, 2010 (month is 0-based)
                Date d2 = c.getTime();
                Locale locIT = new Locale("it", "IT"); // Italy
                Locale locPT = new Locale("pt"); // Portugal
                Locale locBR = new Locale("pt", "BR"); // Brazil
                Locale locIN = new Locale("hi", "IN"); // India
                Locale locJA = new Locale("ja"); // Japan
                DateFormat dfUS = DateFormat.getInstance();
                out.println("US " + dfUS.format(d2));
                DateFormat dfUSfull = DateFormat.getDateInstance(DateFormat.FULL);
                out.println("US full " + dfUSfull.format(d2));
                DateFormat dfIT = DateFormat.getDateInstance(DateFormat.FULL, locIT);
                out.println("Italy " + dfIT.format(d2));
                DateFormat dfPT = DateFormat.getDateInstance(DateFormat.FULL, locPT);
                out.println("Portugal " + dfPT.format(d2));
                DateFormat dfBR = DateFormat.getDateInstance(DateFormat.FULL, locBR);
                out.println("Brazil " + dfBR.format(d2));
                DateFormat dfIN = DateFormat.getDateInstance(DateFormat.FULL, locIN);
                out.println(locIN.getDisplayCountry() + " " + dfIN.format(d2));
                DateFormat dfJA = DateFormat.getDateInstance(DateFormat.FULL, locJA);
                out.println("Japan " + dfJA.format(d2));
        }

        public static void doNumberFormatOperation()
        {
                float f1 = 123.45678f;
                Locale locFR = new Locale("fr"); // France

                NumberFormat[] nfa = new NumberFormat[4];
                nfa[0] = NumberFormat.getInstance(); // max fraction digit = 3 (default)
                nfa[1] = NumberFormat.getInstance(locFR);
                nfa[2] = NumberFormat.getCurrencyInstance(); // max fraction digit = 2 (default)
                nfa[3] = NumberFormat.getCurrencyInstance(locFR);

                for(NumberFormat nf : nfa) {
                        out.print(nf.getMaximumFractionDigits() + " " + nf.format(f1));
                        nf.setMaximumFractionDigits(5);
                        out.println(" " + nf.format(f1));
                }
        }
}
```

And following is the output:
Current Date and Time: Fri **May 05** 11:34:23 IST **2006**

Calendar add() and roll()….
Incrementing date by 30 days...
Sunday, **June 4, 2006**
Rolling Date by another 28 days...
Friday, **June 2, 2006 – year and month were not changed.**

6/2/06

parsed = Fri Jun 02 00:00:00 IST 2006 **– date parsed from string using DateFormat class.**

Checking every part of a given date and time using calendar…
Current Time
ERA: 1
YEAR: 2006
MONTH: 4
WEEK_OF_YEAR: 18
WEEK_OF_MONTH: 1
DATE: 5
DAY_OF_MONTH: 5
DAY_OF_YEAR: 125
DAY_OF_WEEK: 6
DAY_OF_WEEK_IN_MONTH: 1
AM_PM: 1
HOUR: 0
HOUR_OF_DAY: 12
MINUTE: 34
SECOND: 23
MILLISECOND: 173
ZONE_OFFSET: 5
DST_OFFSET: 1

Current Time, with hour reset to 3
ERA: 1
YEAR: 2006
MONTH: 4
WEEK_OF_YEAR: 18
WEEK_OF_MONTH: 1
DATE: 5
DAY_OF_MONTH: 5
DAY_OF_YEAR: 125
DAY_OF_WEEK: 6
DAY_OF_WEEK_IN_MONTH: 1
AM_PM: 1
HOUR: 3
HOUR_OF_DAY: 15
MINUTE: 34
SECOND: 23
MILLISECOND: 173
ZONE_OFFSET: 5
DST_OFFSET: 1

Locale related operations….
US 12/14/10 11:34 AM
US full Tuesday, December 14, 2010
Italy marted∞ 14 dicembre 2010
Portugal Terta-feira, 14 de Dezembro de 2010
Brazil Terta-feira, 14 de Dezembro de 2010
India ???????, ?? ??????, ????
Japan 2010?12?14?

Number formatting….

3 123.457 123.45678
3 123,457 123,45678
2 $123.46 $123.45678
2 123,46 ñ 123,45678 ñ

TABLE 6-3 Instance Creation for Key java.text and java.util Classes

Class	Key Instance Creation Options
util.Date	new Date(); new Date(long millisecondsSince010170);
util.Calendar	Calendar.getInstance(); Calendar.getInstance(Locale);

util.Locale	Locale.getDefault();
	new Locale(String language);
	new Locale(String language, String country);

text.DateFormat	DateFormat.getInstance(); DateFormat.getDateInstance(); DateFormat.getDateInstance(style); DateFormat.getDateInstance(style, Locale);
text.NumberFormat	NumberFormat.getInstance() NumberFormat.getInstance(Locale) NumberFormat.getNumberInstance() NumberFormat.getNumberInstance(Locale) NumberFormat.getCurrencyInstance() NumberFormat.getCurrencyInstance(Locale)

Parsing, Tokenizing, and Formatting (Objective 3.5)

- **regex** is short for regular expressions, which are the patterns used to search for data within large data sources.
- **regex is a sub-language** that exists in Java and other languages (such as Perl).
- regex lets you to create search patterns using literal characters or metacharacters. Metacharacters allow you to search for slightly more abstract data like "digits" or "whitespace".

`[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)

Note: [ab] defines a set of characters one of which is to be matched.
[a-d[m-p]] is a way to do union ie character to be matched is a thru d or m thru p.
[a-z && [def]] is a way to do intersection of sets of characters ie d,e or f is to be matched.
[a-z&&[^bc]] is a way to do subtraction of sets of chars ie a thru z except b and c to be matched.

- Study the `\d`, `\s`, `\w`, and . metacharacters

Metacharacters:

· Any character

`\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]`

- regex provides for ***quantifiers*** which allow you to specify concepts like: "look for one or more digits in a row."
- Study the `?`, `*`, and `+` greedy quantifiers.

Greedy quantifiers

X?	*X*, once or not at all
*X**	*X*, zero or more times
X+	*X*, one or more times
X{*n*}	*X*, exactly *n* times
X{*n*,}	*X*, at least *n* times
X{*n*,*m*}	*X*, at least *n* but not more than *m* times

When you use the *, +, and ? quantifiers, you can fine tune them a bit to produce behavior that's known as "greedy," "reluctant," or "possessive."

`?` is greedy, `??` is reluctant, for zero or once
`*` is greedy, `*?` is reluctant, for zero or more
`+` is greedy, `+?` is reluctant, for one or more

```
source: yyxxxyxx
pattern: .*xx
```

is using the greedy quantifier * and produces

```
0 yyxxxyxx
```

If we change the pattern to

```
source: yyxxxyxx
pattern: .*?xx
```

we're now using the **reluctant** qualifier *?, and we get the following:

```
0 yyxx
4 xyxx
```

The **greedy** quantifier does in fact read the entire source data, and then it works backwards (from the right) until it finds the rightmost match. At that point, it includes everything from earlier in the source data up to and including the data that is part of the rightmost match.

- Remember that metacharacters and Strings don't mix well unless you remember to "escape" them properly. For instance **`String s = "\\d";`**
- The Pattern and Matcher classes have Java's most powerful regex capabilities.
- You should understand the Pattern **`compile()`**, **`matcher()`** method and the Matcher **`matches()`**, **`pattern()`**, **`find()`**, **`start()`**, and **`group()`** methods.

import java.util.regex.*;

```
import static java.lang.System.*;

public class RegexTest
{
    public static void main(String[] args)
    {
        // regex pattern definition - 0 or 1 chars
        // then 1 or more whitespace then 0 or more a's then 1 to 3 b's
        // then any 1 or 0 char from a thru z except chars from m thru u.
        Pattern p = Pattern.compile(".?\\s+a*b{1,3}[a-z&&[^m-u]]?");

        // associate a data source to perform pattern matching at.
        Matcher m = p.matcher("qq aaaaaabddttssseez aaaaaabbttsss z    aaaaaabbb");

        // perform pattern matching - use if u have just one match to find
        //boolean b = m.matches();

        Pattern p1 = m.pattern(); // gives back the pattern used to match.

        // if no match was found, then
        // a runtime exception raised - IllegalStateException
        while (m.find()) {
            out.println("Sub sequence matched in the source: " + m.group());
            out.println("Start index of last match = " + m.start());
            out.println("End index of last match = " + m.end());
        }
    }
}
```

And the o/p is:

```
Sub sequence matched in the source: q aaaaaabd
Start index of last match = 1
End index of last match = 11
Sub sequence matched in the source: z aaaaaabb
Start index of last match = 19
End index of last match = 29
Sub sequence matched in the source: z    aaaaaabbb
Start index of last match = 35
End index of last match = 49
```

- You WON'T need to understand Matcher's replacement-oriented methods.
- You can use **java.util.Scanner** to do simple regex searches, but it is primarily intended for tokenizing.

```
// read an int from input - uses default delimited whitespace
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();

// long types to be assigned from entries in file myNumbers
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```

```
// use of regex in specifying the delimiter
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();
```

o/p is:

```
1
2
red
blue
```

```
Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L").useDelimiter(" ");
while(sc.hasNext()) {
	if(sc.hasNextInt())
		System.out.print(sc.nextInt() + " ");
	else sc.next();  // advance to the next token
}
```

If sc.next() is not used then the while loop will not progress beyond the first non-int data and loop endlessly.

- Tokenizing is the process of splitting delimited data into small pieces.
- In tokenizing, the data you want is called tokens, and the strings that separate the tokens are called delimiters.
- Tokenizing can be done with the Scanner class, or with `String.split()`.
- Delimiters are single characters like commas, or complex regex expressions.
- **The Scanner class allows you to tokenize data from within a loop, which allows you to stop whenever you want to.**
- **The Scanner class allows you to tokenize Strings or streams or files.**
- **The `String.split()` method tokenizes the entire source data all at once, so large amounts of data can be quite slow to process.**

```
import java.util.*;
class SplitTest {
	public static void main(String[] args) {
		String[] tokens = args[0].split(args[1]);
		System.out.println("count " + tokens.length);
		for(String s : tokens)
			System.out.println(">" + s + "<");
	}
}

java SplitTest "aaaaas2 sss43sss" "\d"

O/P is:

count 4
>aaaaas<
> sss<
><
>sss<
```

- **New to Java 5** are two methods used to format data for output. These methods are **`format()`** and **`printf()`**. These methods are found in the **PrintStream** class, an instance of which is the `out` in **`System.out`**. These two methods behave exactly the same way and take exactly same arguments and return same output.