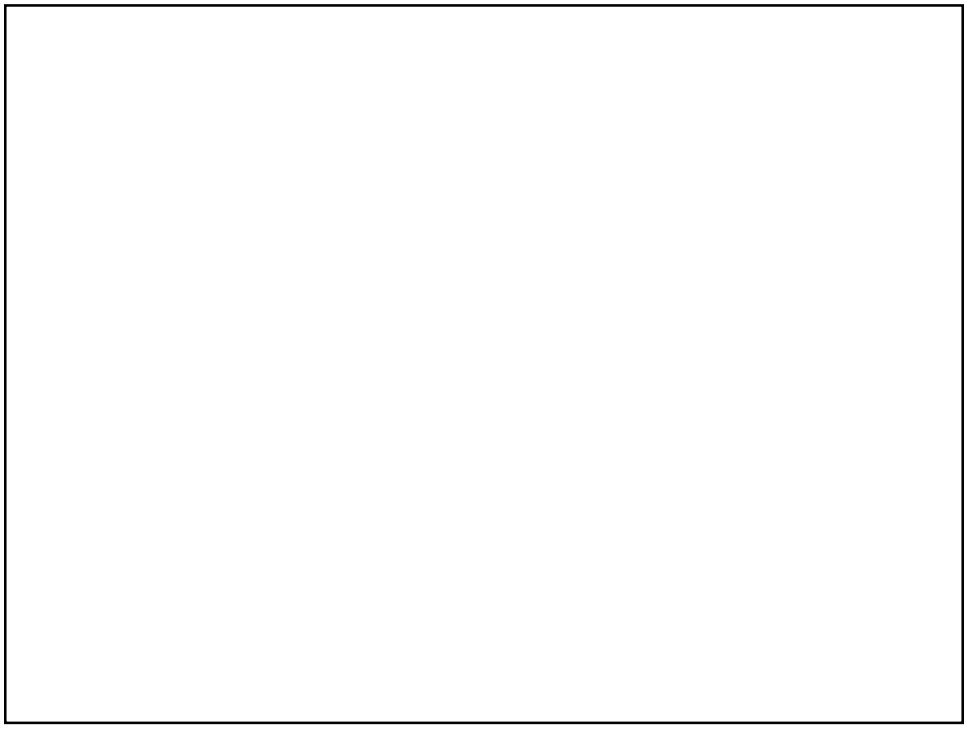


IBM System i™

Session: 480063 Agenda Key: 23MN

Java™ 101: Basic Syntax and Structure

Osman Omer
(oomer@us.ibm.com)

i want stress-free IT.
i want control.
i want an i.

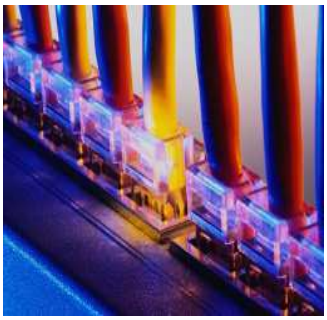
Outline

- Introduction
- Why Java™?
- Object-Oriented Overview
- Java Keywords and Definitions
- Elementary Java Structure
- Java Syntax and Control Flow
- Compiling and executing Java Code
- Tips for Approaching Java Code
- Tools for Java Development

Introduction

- Goals
  - Introduce basic Java syntax elements
  - Compare control flow constructs with those in RPG
  - Develop skills for reading and understanding Java source code
- Expand skills in writing Java code
  - Get you understanding Java code syntax
  - Help you find different ways of looking at code
- How to get there
  - Look at Java code
  - Help you understand it

Why “Java”? It’s simple.

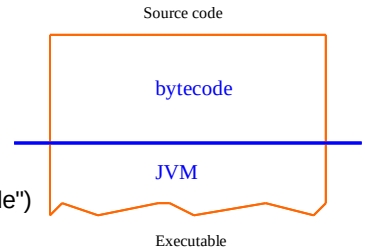


- Most popular language
- Distributed
- Secure
- Robust
- Multithreaded
- Write once, run anywhere
- Internationalization (I18N)

OO in 5 Minutes or Less

- **Class**
  - A pattern, template, or model for an **object**
  - Existence is independent of any single VM
    - Stored ‘externally’ in files in the filesystem
    - Classes can define “inner” classes since Java2
  - Invariant (constant) *class* data shareable
- **Object**
  - Defn: combination of **data** and **methods**
  - An **instance** of a class
  - Existence *depends* on a VM to “hold” the object
- **Data**
  - The **fields** of an object (attributes, characteristics)
- **Methods**
  - The **functions** of an object (procedures, subroutines)

Java Definitions



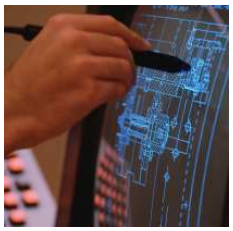
- **Classfile**
  - A file in the hierarchical file system
  - Contains Java "object code" (a.k.a "bytecode")
  - Result of compiling Java **source code**
- **Jar File**
  - **J**ava **AR**chive; a **collection** (zipfile) of classfiles and other resources
- **Virtual Machine for Java (VM)**
  - Software that loads and executes **bytecode**
- **Classpath**
  - Locations searched by the VM for classes and other resources
- **Package**
  - Collection of related classes
  - Provides access **protection** and name space management

Java Keywords

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

A few minutes on Java and “code reuse”

- Writing code, such as Java, blends two orientations:
  - Coding ***FROM* reuse** (using code that others provide)
  - Coding ***FOR* reuse** (writing code to provide to others)
- ***FROM* reuse**
  - *Everybody* codes "from reuse" in Java
    - …every time you use a java.lang.String, for instance
  - Typically, “applications” are *strictly FROM* reuse
- ***FOR* reuse**
  - Designing classes to be used by others (i.e. API)
    - Up-front design work fundamentally important
    - Migration paths, versioning, encapsulation
  - Definitely “advanced” Java programming
- **We will stick to “FROM reuse” initially**



Caveats

- We only have 1¼ hours total – just time for the "high points"
  - The complete Java language "specification" (JLS) is available here:
    - http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
  - This specification includes a complete "grammar" for Java
    - Formal specification of keywords and their valid relationships
- We will focus mainly on traditional Java classes
  - Class, method and field declaration syntax
  - Method bodies and control flows
  - No fancy stuff (inner classes, abstract classes)
- For further study, see The Java Tutorial
  - A very valuable source of do-it-yourself instructional materials
  - http://java.sun.com/docs/books/tutorial/index.html

Gross anatomy of a Java source file

- **Comments**
  - Comment delimiters modeled on C++
  - Can go anywhere – use liberally
  - Special "javadoc" comments
    - Self-documenting code – tied to what *follows*
    - Absolutely VITAL when coding "FOR reuse"
- **Package statement**
  - Provides "namespace" for declared types
  - Must be **first non-commentary statement**
  - Implies **directory structure** for source code
- **Import statements**
  - Provides a way to include groups of classes
  - Declares the "domain of reuse"
  - NOTE: **`java.lang.*`** – no import required
- **Type(s)**
  - Typically one class (or interface) per file
  - Aways the same name as the file
  - Enumerated types (new in JDK 1.5)

```
/* File: com/ibm/examples/HiThere.java
 * Provided as-is, a simple Java example.
 *   Usage: java com.ibm.examples.HiThere
 */
package com.ibm.examples;

/* Import to get java.sql.Timestamp */
import java.sql.*;

/** Very simple example application,
 * with just one method.
 */
public class HiThere {

  /* Every application has a "main" */
  public static void main(String[] args) {
    long now = System.currentTimeMillis();
    Timestamp ts = new Timestamp(now);
    System.out.println("Time is: " + ts);
  } // end of main()

} // end of HiThere
```


It's all there -- between the curly braces

- Remember:
  - **Java classes define the methods and fields of an object**
- Class declaration is itself delimited by curly braces { }
  - Java's C/C++ heritage shows through
    - Not the last time we'll see their importance
- Classes:
  - May declare fields and methods (explicitly within the braces)
  - Will inherit any fields and methods of their superclasses
- It is easy to tell a method() apart from a field:
  - Method declarations ALWAYS have parentheses (empty or not)
  - Field declarations do NOT need parentheses (there are no parameters!)
  - Note that some methods do NOT have a body!
    - (i.e. `abstract` and `native` do not specify a method body)

Fields and methods

```
class ExampleClass extends SomeSuperClass
{

    SomeType dealyBobber;            // Field or method?
    static SomeType thingamaJig;     // Field or method?

    SomeType doHickey()              // Field or method?



    static SomeType whatzItz(int i)  // Field or method?



} // End of ExampleClass
```


Fields and methods

```
class ExampleClass extends SomeSuperClass
{

    SomeType dealyBobber;            // Field or method?
    static SomeType thingamaJig;     // Field or method?

    SomeType doHickey()              // Field or method?
    {
        return this.instanceField;
    }
    static SomeType whatzItz(int i)  // Field or method?
    {
        return staticField;
    }

} // End of ExampleClass
```

Static vs. instance fields and methods

- What does 'static' mean?
  - While a class is a "template" for an object, it's also an object itself!
  - Static methods and fields are scoped with the class, not with any instance

- Example: keep a count of all cars in this VM (extra credit for finding the bug…)

```
class Car {
  static int numCars;  // static field to hold count of cars
  String myMake;       // instance's make
  String myModel;      // instance's model
  int    myModelYear;  // instance's year

  // special method, called a "constructor"
  public Car(String make, String model, int modelYear) {
    myMake = make; myModel=model; myModelYear = modelYear;
    // In the constructor, increment the static number of cars
    numCars++;
  } // end of constructor
  public setCarInfo(Car(String make, String model, int modelYear) {
    myMake = make; myModel=model; myModelYear = modelYear;
  } // end of setCarInfo
} // end of Car class
```


Static vs. instance fields and methods

- A CarDealer application to utilize the Car class

```
public class CarDealer {
  public static void main(String[] args){
    Car a1("Chevy", "Cavalier", 2001);
    Car b3("Dodge", "Dynasty", 1996);
    Car a2("Nissan", "Ultima", 2006);

    b3.setCarInfo("Hammer", "H3", 2004);
    int num = Car.numCars; //notice the class name qualifier
  } // end of main method
} // end of CarDealer class
```

Class Declaration Syntax

- [modifier*] class <ClassName> [extends <AnotherClass>] [implements <InterfaceName*>]
- Class modifiers:
  - public: can be accessed from outside the package;
  - abstract: can not instantiate; usually has one or more abstract method; and
  - final: can not be subclassed.
- Examples:
  - class File
  - final class Student extends Person
  - public abstract class Farm implements Land, House, Animal

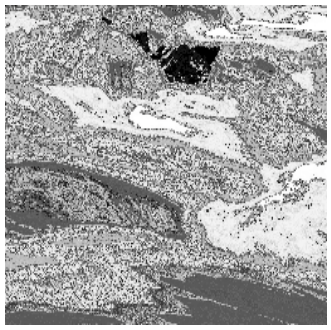
Field Declaration Syntax

- [modifier*] <TypeName> <fieldName> [= initializerValue];
- Field modifiers:
  - public: accessible wherever the class name is accessible
  - protected: accessible to subclasses and all classes in the same package
  - private: only accessible to the class it declares it
  - static: field is associated with the class not the object. One copy for the class and shared among all objects.
  - final: Once set, value can not be changed (immutable)
- TypeName: one of the 8 primitive data types or any non-abstract class
- Examples:
  - boolean isSenior;
  - static final int WHITE=1;//declared in the Color class and access Color.WHITE;
  - public String name = file.getName();

Method Declaration Syntax

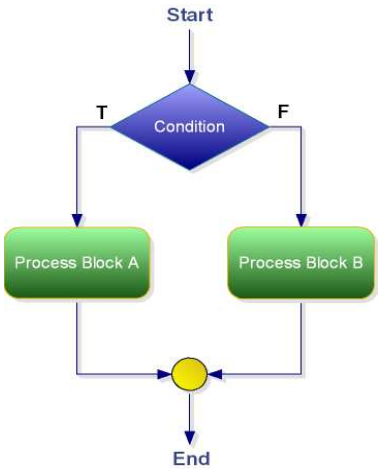
- [modifier*] <returnType> <methodName>([<args>]) [throws <ExceptionName*>]
- Field modifiers:
  - All those of the field (with final meaning can not be overridden) plus
  - abstract: no code, part of abstract class, subclass must implement or re-declare as abstract
  - synchronized: for locking/unlocking by threads; and
  - native: field is associated with the class not the object
  - final: Once set, value can not be changed (immutable)
- returnType: one of the 8 primitive data types or any non-abstract class
- Examples:
  - public static void main(String [] args)
  - private final long calculateArea()
  - public String getName()

Control Flow Constructs



- Conditional
  - Branches and loops
- Exceptional
  - try, catch, throw
- Unconditional
  - Method calls

Conditional Control Flow: if/else



- **`if/else`**
  - executes block if expression evaluates to “true”

Conditional Control Flow: if/else

Java Syntax

```
if (nameLength > 8) {
  truncate = true;
} else {
  truncate = false;
}
```

```
ticketPrice = 10;
If(age > 65){
  ticketPrice -= 2;
}
```

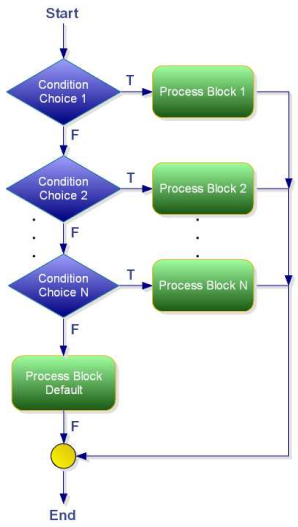
RPG Syntax

```
if nameLength > 8;
     truncate = *ON;
else;
     truncate = *OFF;
endif;
```

```
ticketPrice = 10;
if age > 65;
  ticketPrice = ticketPrice – 2;
```

Conditional Control Flow: switch

- **`switch`**
  - An easy-to-read collection of if statements
  - Use the **`break`** keyword to transfer control to just after the switch statement
  - Argument to **`switch()`** must be scalar value (i.e. integer or character)



Conditional Control Flow: switch

Java Syntax

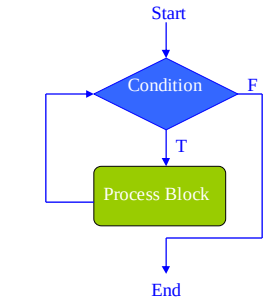
```
switch (status) {
  case 1:
    System.out.println (“Error.”);
    break;
  case 2:
    System.out.println (“End of File.”);
    break;
  default:
    System.out.println (“Success!”);
    break;
}
```

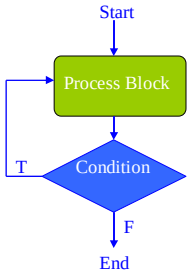
RPG Syntax

```
select;
 when status = 1;
   dsply ‘Error.’;
 when status = 2;
   dsply ‘End of File.’;
 other;
   dsply ‘Success!’;
endsl;
```

Conditional Control Flow: Loops

- **while**
  - **while (condition) { statements }**
- **do … while**
  - **do { statements } while (condition) ;**
- **for**
  - **for (init; condition; incr) { statements }**





Conditional Control Flow: Loops

Java Syntax

```
while(!eof(file)) {
  processRecord();
}

do {
  processRecord();
} while(x > array.length);

for(i = start; i < end;
    i += inc) {
  processRecord();
}
```

RPG Syntax

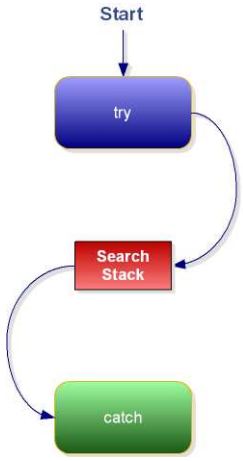
```
dow not %eof(file);
  processRecord();
enddo;

dou x > %elem(Array);
  processRecord();
enddo;

for i = start by inc
    to %elem(array);
  processRecord();
endfor;
```

Exceptional Control Flow

- **`try/catch`**
  - Errors are propagated up the stack
  - Always list catch blocks from most specific to most general
  - The **finally** statement is always executed
  - Must re-throw exception using the **throw** keyword if not handled



Exceptional Control Flow

Java Syntax

```
try {
   // code that might 'throw'
} catch(FileException e1) {
   // handle file error
} catch(Exception e2) {
   // handle all other errors
} finally {
   // ALWAYS do…
}
```

RPG Syntax

```
MONITOR
   // code that might 'throw'
ON-ERROR *FILE
   // handle file error
ON-ERROR
   // handle all other errors
ENDMOD
```

Unconditional Control Flow: Method Calls

```
import javax.swing.*;

class MyWhy {
  public static void main( String[] args ) {
     String question = "Why \"i\"?";
     String answer = showDialog("Question!", question);
     System.out.println( question + " " + answer );
     System.exit(0);
  }

  static String showDialog(String title, String message) {
    String out = JOptionPane.showInputDialog(
         new JFrame(),
         message,
         title,
         JOptionPane.PLAIN_MESSAGE);
    return out;
  }
} // end of MyWhy
```


Be Cautious as You Program in Java

- A Java code file must contain one and only one (outer) class with possible inner classes.
- The file name must match exactly the name of the (outer) class it declares.
- Java is very strictly case sensitive.

Compiling and Executing Java Code

- A JDK must be installed for source compilation.
- From a command line, the Sun compiler command is
  - javac myApp.java
- This command produces a classfile named myApp.class.

- A JRE (JVM) must be installed for execution of the bytecode in the myApp.class
- The Sun execution command is
  - Java myApp

Tips for Approaching Java Code

- Check packaging
  - Jar or classes
  - Does it include the source?

- API or Application
  - If it is an API, evaluate the interface
  - If it is an Application, look for main and run it

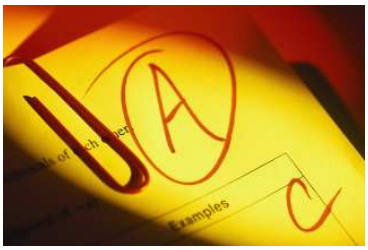
- Look at documentation
  - Is there a javadoc? (Should be for API!)

Tools for Java Development

- Development Environments and Editors
  - Eclipse
  - WebSphere Studio Application Developer
  - jEdit
- Modeling
  - Unified Modeling Language (UML)
  - Rational Rose XDE
- Decompilers
  - DJ Java Decompiler
  - jShrink

Summary

- Scratching the surface…
  - Why Java?
  - Terminology
  - Java Structure and Syntax
  - Development Tools and Tips



- Java Resources
  - Sun's Java Website (java.sun.com)
  - IBM developerWorks
  - IBM Toolbox for Java

Example Application "Mydu.java" (pg. 1)

Usage: Mydu [-a | -s] [-k | -b] [filename …]

```
// file: Mydu.java
import java.io.*;
import java.util.*;

public class Mydu {

    private int showDivisor = 1;        // handles '-b' (blocks) or '-k' (kbytes) flags
    private char showChar = 'd';        // handles '-a' (all) or '-s' (summary) flags
    private boolean verbose = false;    // if true, extra output is generated
    private boolean help = false;       // if true, just writes message and dies
    private int depth = 0;              // recursion depth for indented output
    private ArrayList<String> cmdArgs;  // holder for filenames passed on command line

    public static void main(String[] args) {
        Mydu me = new Mydu(args);
        me.info("Begin processing...");
        me.perform();
        me.info("Processing complete.");
    }

    // constructor (<init>)
    public Mydu(String[] args) { cmdArgs = parseArgsRemoveFlags(args); }

    private void report(long len, String rptName) {
        System.out.printf("%10d %s%n", ((len + (showDivisor-1)) / showDivisor), rptName);
    }
```


Example Application "Mydu.java" (pg. 2)

```
    public void perform() {
        for (String s : cmdArgs) {
            info("...processing command line arg '" + s + "'");
            File f = new File(s);
            if (f.exists()) {
                if (f.isDirectory()) {
                    long thisLen = recurse(f);
                    if (showChar == 's') report(thisLen, s);
                } else {
                    report(f.length(), s);
                }
            } else {
                warn("Cannot find '" + s + "' not a file or directory");
            }
        }
    }

    private void warn(String s) { System.err.println("WARNING:" + s); }
    private void info(String s) { if (verbose) System.err.println("INFO:" + s); }
    private void info_indent(int indent, String msg) {
        if (verbose) {
            if (indent <= 0) indent = 1;  // Internal error, actually
            System.err.printf("%" + indent + "s%s%n", "", msg);
        }
    }
```

Example Application "Mydu.java" (pg. 3)

```
    private long recurse(File inDir) {
        long accumulator = inDir.length();
        try {
            info_indent(++depth, "> ENTER:" + inDir);
            String outerName = inDir.getCanonicalPath();
            // run the directory listing of 'inDir', recursing into subdirs
            for (String wrkName : inDir.list()) {
                File subDir = new File(outerName + File.separator + wrkName);
                String loopName = subDir.getCanonicalPath();
                if (subDir.isDirectory()) {  // BUG: if subDir a self-symlink
                    accumulator += recurse(subDir);
                } else {
                    accumulator += subDir.length();
                    if (showChar == 'a')  report(subDir.length(), loopName);
                }
            }
            if (showChar != 's') { report(accumulator, outerName); }
            info_indent(depth--, "< LEAVE:" + inDir);

        } catch(IOException ioe) {
            throw new RuntimeException("Error in recurse", ioe);
        } finally {
            return accumulator;
        }
    } // end of recurse
```


Example Application "Mydu.java" (pg. 4)

```
    private ArrayList<String> parseArgsRemoveFlags(String[] args) {
        ArrayList<String> rtnVal = new ArrayList<String>();
        for (String a : args) {
            if (a.equals("-h")) {
                System.err.println("Usage: mydu [-k|-b] [-s|-a] [<name>...]");
                System.exit(1);
            } else if (a.equals("-k")) { showDivisor = 1024;
            } else if (a.equals("-b")) { showDivisor = 512;
            } else if (a.equals("-a")) { showChar = 'a';
            } else if (a.equals("-s")) { showChar = 's';
            } else if (a.equals("-v")) { verbose = true;
            } else {
                rtnVal.add(a);
            }
        }
        if (rtnVal.isEmpty()) rtnVal.add(".");
        return rtnVal;
    }

} // end of class Mydu
```

References

- All things Java: http://java.sun.com
- developerWorks: http://www-130.ibm.com/developerworks/
- IBM Toolbox for Java: http://www-1.ibm.com/servers/eserver/iseries/toolbox/
- eclipse: http://www.eclipse.org
- WebSphere: http://www-306.ibm.com/software/websphere/
- jEdit: http://www.jedit.org
- UML: http://www.uml.org
- Rational: http://www-306.ibm.com/software/rational/
- DJ Java Decompiler: http://members.fortunecity.com/neshkov/dj.html
- Jshrink: http://www.e-t.com/jshrink.html
- The Java Tutorials: http://java.sun.com/docs/books/tutorial/index.html

Trademarks and Disclaimers

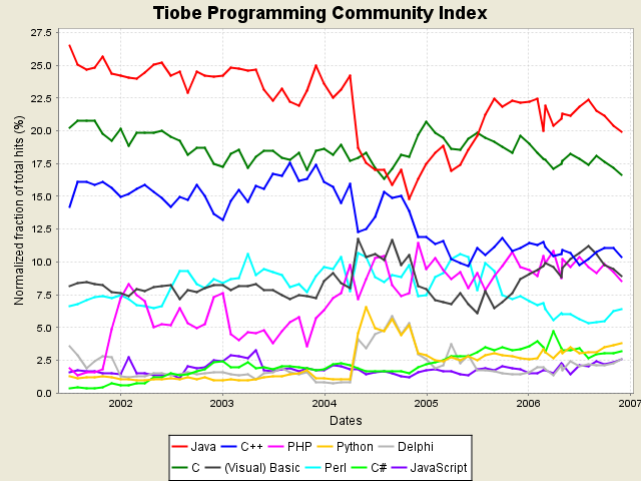
Backup Slides

Programming Languages Popularity

Position Dec 2006	Position Dec 2005	Delta in Position	Programming Language	Ratings Dec 2006	Delta Dec 2005	Status
1	1	=	Java	19.907%	-2.36%	A
2	2	=	C	16.616%	-1.75%	A
3	3	=	C++	10.409%	-0.39%	A
4	5	↑	(Visual) Basic	8.912%	+1.33%	A
5	4	↓	PHP	8.537%	-2.24%	A
6	6	=	Perl	6.396%	-0.74%	A
7	8	↑	Python	3.762%	+1.00%	A
8	7	↓	C#	3.171%	-0.11%	A
9	10	↑	Delphi	2.569%	+1.11%	A
10	9	↓	JavaScript	2.562%	+0.68%	A

Source: TIOBE Programming Index for Dec 2006 - http://www.tiobe.com/

Programming Languages Popularity



Source: TIOBE Programming Index for Dec 2006 - http://www.tiobe.com/

javadoc

```
java/lang/Object.java
/**
 * Class <code>Object</code>
 * Every class has <code>Obj
 * . . .
```

javadoc