

Java

Language basics

- Just another language from the “algol family”
 - Data types
 - Variables (and constants)
 - Operators
 - Expressions (same as always)
 - Control flow
 - Functions

Algol language family

Algol - AlgolW, Pascal, Modula2, Ada, Oberon, ...
- CPL, BCPL, B, C
- Simula67
- Algol68

```
graph LR; C --> Java; Cplusplus[C++] --> Java; Eiffel --> Java;
```

Algol, 1960, initially conceived of as language for defining mathematical algorithms; implementations appeared by 1962

AlgolW (~1968), Pascal (~1970), Modula2 (~1978), Oberon (~1980s) – N. Wirth’s family of languages primarily aimed at teaching programming

Algol language family

Algol - AlgolW, Pascal, Modula2, Ada, Oberon, ...
- CPL, BCPL, B, C
- Simula67
- Algol68

```
graph LR; C --> Java; Cplusplus[C++] --> Java; Eiffel --> Java;
```

CPL – overambitious “Combined Programming Language” proposal (early 1960s) with Algol like features and some business data processing features (akin to IBM’s PL/1)
BCPL – tiny subset of CPL, easily compiled to efficient code, aimed really at “systems programming tasks” (op.sys, utilities like editors, etc)
C – reworked BCPL

All languages in family have strong similarities

- Basic “built in” (“primitive”) data types
- Similar expressions
- Similar flow control within a function
 - Sequence
 - Selection
 - Iteration
- Similar function call mechanisms
- Once you’ve learnt one, you have good basis for switching to using another

Data types

- A few “built in” (primitive) types, everything else will be an instance of a class
- *Unlike C++, you can’t make your classes “first class types” comparable in all respects to these “built in” types.*
- Built in types (aka “primitive” types)
 - integer arithmetic
 - characters
 - real arithmetic
 - boolean

boolean

- boolean true false
- *boolean is not compatible with integer types*
- Significant tidy up from C where have automatic conversion of integer type to/from boolean (0 false, non-zero true), an automatic conversion that is often source of errors ---
 - $(0.5 < x \leq 1.0)$ in C++ is legal but does not check whether x in range $(0.5, 1.0)$; expression is illegal in Java ($0.5 < x$ is boolean value, can't compare boolean \leq real)

Built-in types – mainly for arithmetic

- Mainly for efficient arithmetic processing.
- Will later encounter class types that represent numerical values –
 - Integer (int value)
 - Float (float value)
 - Double (double value)
- Not used for arithmetic processing, used when want to package a numeric value as an object (Why? Wait – examples later).

Enumerated type

- C++ enum

```
enum grade { Fail, PC, P, CR, D, HD };
```

 Fail etc are values of type grade. Can define grade variables etc
- Enumerated types introduced in Java 1.5

```
public enum StopLight { red, amber, green };
```
- (*StopLight would be a member of a class that uses this enum – public if enum to be used by clients of class*)
- Role similar to enums in C++ (but cannot do things like associate integer values with enum values as can do in C++)

Fancier Java 1.5 enum

- *“Java programming language enums are far more powerful than their counterparts in other languages, which are little more than glorified integers. The new enum declaration defines a full-fledged class (dubbed an enum type). It allows you to add arbitrary methods and fields to an enum type, to implement arbitrary interfaces, and more.”*

Sun example of fancy enum:

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS (4.869e+24, 6.0518e6),
    ...
    PLUTO (1.27e+22, 1.137e6);
    private final double mass;
    private final double radius;
    Planet(double mass, double radius)
    { this.mass = mass; this.radius = radius; }
    public double mass() { return mass; }
    public double radius() { return radius; }
    ...
}
```



Variables

- Different treatment of primitives and class instances
 - primitive (built in) types on stack
 - class instances on heap (free store)
- (In C++, don't have this restriction).
- No global (or filescope) variables of any type.

Variable names

- Variable name must start with a “letter”
 - any letter from any Unicode alphabet, but stick to a..z, A..Z, _, \$
- Variable name can contain letters, digits

Variable declarations

- Variables can be declared as -
 - instance data members (each object created from class has its own copy)
 - class data members (one copy of variable shared by all instances of class)
 - local variables of member functions

Variable declarations

- Simple declarations -


```
int      counter;
double   slope, intercept;
char     mark;
```

 - *data members are initialized (zero numerics, false booleans)*
 - *local variables of functions are **not** automatically initialized*

So what did he mean by that? !

```
class Demo {
    private String name; // initialized to null
    private int count; // initialized to 0

    public void someFunction()
    {
        int temp; // uninitialized variable
        String whatever; // Uninitialized variable
    }
}
```

Variable declarations

- Declaration may provide an initial value (should do this for all local variables)


```
double    sumxsq = 0.0;
char      hash = '#';
```



constants

- (The word `const` is a reserved word in Java language; currently it is not used.)
- Java has a qualifier “`final`” that can modify declarations
 - can specify a class as `final` (can’t make subclasses)
 - can specify a member function as `final` (subclasses cannot redefine that function)
 - can specify an initialized variable as `final` (initial value cannot be changed, so it is a constant)

© nabg

constants

- Since all data elements must be class members (or local variables in functions), constants are generally defined as final data members of a class.
- Since want only one copy of constant (not one for each instance), will make data element a class member (as in C++, this involves use of static qualifier)

© nabg

constants

- Typical example “PI”
 - in class Math


```
public final static
  double PI = 3.141.....;
```
- order of qualifiers like final and static can be varied

© nabg

constants

- Constant strings are common (these will be declared as members of application specific classes)

```
private static final String dataFilename = "data.txt";
```

© nabg

Constant strings


- java.lang.String
 - holds a constant String
 - Less functionality than string class that you used in C++; does have some member functions
 - check whether a String contains a substring
 - Create new String with all lower case letters
 - ...
 - Can contain Unicode characters (defined by \u escape combinations)
 - Characters like “ and \
 - Have to be “escaped” as in C++ strings “\” “\\”
 - Many operations on String data elements create new String objects allocated in heap (sometimes expensive)

© nabg

“Derived types”? No, not in Java

- In C/C++ you can declare variables with “*derived types*” like **pointer** types, **arrays**, **references**.
- You can’t do that in Java.
- Arrays will come soon, they are considered as being much the same as objects (class instances)

© nabg



operators

- Generally similar to those in C/C++.
- No operators relating to pointer types.
- An extra shift operator to resolve ambiguities that causes portability problems for C/C++ programs.

© nabg

Operator precedence table

[] array subscripting, . member access, () function call
 ! logical not, ~ bit not, ++, --,
 * / %
 + -
 << >> >>>
 < <= > >=
 = = !=
 &
 ^
 |
 &&
 ||
 ?:
 = += -= etc

Use parentheses on any expression complex enough to make you concerned about operator precedence!

© nabg

Shift operators

- << shift left $n \ll 3$
- >> “arithmetic shift right”
- >>> “logical shift right”
- C/C++ leave meaning of >> undefined, compiler writer can chose either arithmetic or logical right shift (hence portability problems!).

© nabg

operators

- operators are essentially only defined for built in (primitive types)
- Unlike C++, you can not define a ++ operator to work with instance of your classes (so no class Point where ++() operator function does vector addition)
- One exception, Java did define operator+() for class String; can use + with String variables, it produces a new String by concatenating existing Strings (this operator will do type coercion, String + number invokes function to convert the number to a String).


© nabg

operators

- Reminder on ++ and --

x= 10;	x= 10;
y=++x;	y=x++;
Now x=11; y=11;	Now x=11; y=10;

© nabg



Control flow

- The usual -
 - selection
 - if
 - if ... else ...
 - switch
 - iteration
 - while
 - do ... while (repeat)
 - for(...;...;...)
 - better to use “Enumerators” (iterators) when working through collections

© nabg

Control flow - loops

- while with break & continue
 - Same as C++, break takes you out of a loop, continue takes you to loop-control test
- for – counting style (for int i=0;i<lim;i++)
 - Same as C++, break takes you out of a loop, continue takes you to loop-control test
- Java 1.5 modified for
 - Perl inspired “for each” loop, used for working through a collection (examples later)
 - “for each element of collection do ...”

```

© nabg

int[] data = { 13, 24, 35, 36, 42, 51, 54, 56, 57,
              68, 71, 82, 86, 89, 90, 92, 93 };
System.out.println("All of them");
for(int i=0;i<data.length;i++)
    System.out.println(data[i]);

System.out.println("Odd ones");
for(int i=0;i<data.length;i++)
{
    if((data[i] % 2) == 0) continue;
    System.out.println(data[i]);
}
System.out.println("Even ones");
for(int item : data)
{
    if((item % 2) == 1) continue;
    System.out.println(item);
}

```

"for-each" loop

Control flow


- Some oddities
 - “labels” and modified forms of “break” and “continue” (keyword `goto` is reserved but not used)

```

getRowData:
for(int row=1; row<=nRows; row++) {
    ...
    if(!ValidateDataSoFar(row)){
        System.out.println("data inconsistent!");
        continue getRowData; }
    ..
}

```

Don't know why they left this historical oddity in the language.



Functions

- Form is generally similar to C/C++.
- But
 - all functions must be declared as member functions of a class (possibly static member functions eg. `Math.sin(double)`, etc)

Functions

```

int largest(int data[], int num)
{
    int largest = data[0];
    for(int i=1;i<num; i++)
        if(data[i]>largest) largest = data[i];
    return largest;
}

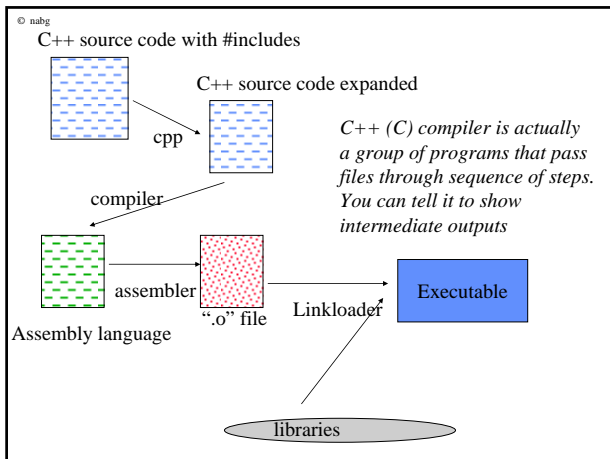
```

- void functions similar to C/C++

Program structure,
 compilation,
 class loading,
 ...

Program structure

- Superficial similarities to C/C++, but underneath there are significant differences.
- C & C++ build program by compiling source files to `.o` ('object') files and then linking these with libraries to get an executable.



© nabg

Java compile time

- With Java
 - a file will contain declaration of a principal class (and optionally some auxiliary classes)
 - all details of class in the one file (unlike C++, no separate header file, and no possibility of defining some member functions in one file and others in a different file)
 - name of file must match name of (principal) class (with file extension .java)
 - compiler creates files with byte codes (separate '.class' file for each class defined in the file being compiled)

© nabg

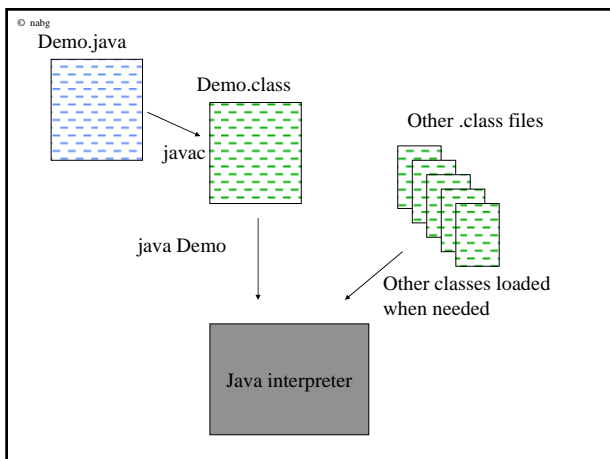
Compiler

- A '.java' file will start with "import" statements
 - role has some similarities with #includes in C/C++
 - if you are going to be declaring and using instances of classes defined elsewhere, compiler is going to have to read descriptions of those classes so that it can verify that they support the operations that you invoke on the instances that you create

© nabg

Java runtime

- Two possibilities
 - simple applications run by the 'java' interpreter
 - applets (meant to be run by a browser that includes a Java interpreter)
- Either way,
 - byte codes of 'principal' class get loaded
 - "appropriate function" called, normally this will create one (or more) objects and get one to run
 - code for other classes loaded
- Java interpreter loads code for classes when it needs to, there is no real "link" step.



© nabg

Runtime

- Runtime system must 'know' where byte codes for other classes can be found
 - current directory
 - default directory (for classes supplied as part of standard Java release)
 - directories specified by CLASSPATH environment variable
 - possibly, loaded from remote site via Internet

Classpath ... load-library path

- C++ does face same issue
 - Where are the libraries like iostream, stdio, etc?
- Usually defined by “environment” variables that are set when you log in to Unix – try the “env” command on Unix terminal
- Less common in C++ to want to add something (when did you last use a class from the rogue-wave library?)
- So less common to have to modify load-library path

Applications and Applets

- We will start with “Applications”
 - You have to understand a little bit about OO techniques, creating classes, using inheritance, etc to create Applets
 - Applications require simply “object based” programming style
 - you use objects that are instances of standard classes
 - you specify your own classes and create instances of these
 - Applets aren’t that important in practice!

Application

- Java program ~ C++ programs that you are used to
 - Typically run from command-line (though in CSCI213 we will more often just run the application from within NetBeans development system)
 - Can have command line arguments
 - May use text inputs and outputs like C++ programs
 - May have simple graphical user interface

Application

- In an Application, you must provide the main() function
 - *sometimes* this will create an instance of a class that you’ve defined and get it to start on some “work()” function (name of function will vary!)
 - initial CSCI213 examples are simplified, all processing will be done via function calls from main()

“Hello world” example

Kernighan and Ritchie started fashion for silly “Hello world” programs in their book introducing C

HelloWorldApp.java

- File HelloWorldApp.java

```
public class HelloWorldApp {
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

Reminder: filename must match name of ‘principal’ class

Compile & run

```
$ javac HelloWorldApp.java
$ java HelloWorld
```

HelloWorldApp.java

- No imports?
 - There *is* an **implicit** import statement ---
`import java.lang.*`
 - You never put this in your files, the compiler always scans the “header” information relating to *all* classes provided as part of Java’s standard language “package” (library)
 - Here actually using two classes directly ---
String and System

HelloWorldApp.java

```
public class HelloWorldApp {
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

- We have to have a class (there are no free functions) – class HelloWorldApp
- This class can provide the main() function

HelloWorldApp.java

- main() doesn’t depend on having any actual HelloWorldApp object, so it is naturally a static (class) member function
- main() (and class HelloWorldApp) are “public” - their names are to be known by other parts of any system built using them (in particular, names needed by java interpreter which will have to start execution at beginning of main())

main() *must be* public, static, void and have String[] argument

HelloWorldApp.java

```
public static void main(String[] args)
{
    System.out.println("Hello World!");
}
```

- main() has signature (argument list) similar to that for C/C++ programs on Unix (int main(int argc, char *argv[]);
- A main function may have to pick up command line arguments

HelloWorldApp.java

- The arguments (if any) will be in the array of String objects passed to main();
- main() can find how many entries there are in the array, and process the data.
- Here not interested in command line arguments so ignore the array in body of main()

Note that there are subtle differences between C/C++ and Java regarding which entries on command line will actually be given to the program.

HelloWorldApp.java

```
public static void main(String[] args) {
    System.out.println("Hello World!");
}
```

- Body of main() is simply the request to print something.
- “out”
 - is a static data member of class System
 - is a `PrintStream` object,
 - knows how to print a String on “stdout”

cout, System.out

- Where did cout come from in C++?
- It was declared as a global variable in the `iostream` library, one that is automatically initialized when the library code is loaded.
- `System.out` is similar.
- Java’s class `System` defines a public static data member (`out`) that is initialized when the `System` class code is loaded.

System.out : a PrintStream object

- Class `PrintStream` defines lots of member functions (things that a `PrintStream` object can do)
 - `println()`
 - `flush()`
 - `println(String x)`
 - `print(String x)`
 - `println(double x)`
 - `print(double x)`
 - ...
- ```
System.out.println("Hello World!");
```

## String[] args; String args[];

- Array declarations
  - `[]` as modifier on variable – C++ style;
  - `[]` as modifier on type – more typical Java style
- Both styles legal in Java.
  - `String[]` is somewhat preferable


## String not string

- This seems to irritate those who are happy with their C++
  - Have been using `string` class in C++ for a year or more
  - Now meet `String` class in Java (which has significantly different properties)
  - Keep typing as `string`
  - Keep getting compilation error messages
- Just remember – **ALL** class names in (standard) Java start with a capital letter.

## IDE or Notepad/cmd.exe : 1

- In previous years, we got students to develop using simple tools:
  - Edit source in Notepad or similar simple editor (usually define principal class and any auxiliary classes in same source file)
  - Use `cmd.exe` command line environment
    - Compile with `javac`
    - Run with `java`
- Minimal resources needed

## IDE or Notepad/cmd.exe : 2

- This year we are using NetBeans 
  - More complex
  - More things to learn at start
  - Requires more computer resources (though nothing excessive)
- Advantages?

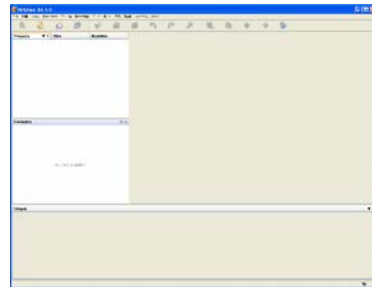
## IDE advantages ...

- syntax aware editor - highlights errors like missing quote marks
- built in copy of the documentation on the classes in the standard Java packages (class libraries)
  - adding the right import statements
  - checking function signatures so as to prompt you to supply the correct arguments.
- Compilation error messages are shown with hyper-text links to the lines with the errors
- Secondary windows provide graphical views of your code - class hierarchies, list of data and function members with their accessibility etc.

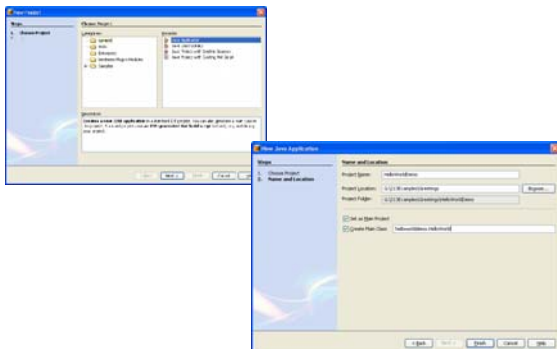
## More IDE advantages ...

- helps organize the various files that make up an application.
- "refactoring" utility - this allows you to do things like move methods from one class to another
- JavaDoc documentation to a class
- test "scaffolding" code
- "source level debugging" ..

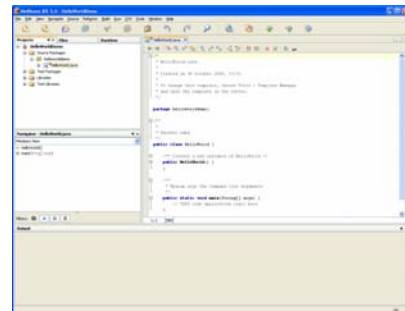
## Start your NetBeans ...



## Create a project



## NetBeans setup ...



## Differences ...



- With NetBeans
  - Easiest if each class goes in a separate file
    - Not required for Java
    - Some of examples in lecture material will show applications with many classes defined in one file
  - “packages”
    - Namespace control
    - Helpful in bigger projects
    - Tiresome if working with Notepad/cmd.exe/javac-java
    - Automated with NetBeans

## NetBeans



- Compile and Run
  - Menu commands
- Data files, command line argument
  - Use “properties” dialog of project to define these

## Back to demo-application

- Use command line arguments ...

```

© nabg
$ java HelloWorldApp Bozo
Hello Bozo

```

## HelloWordApp.java

```

public class HelloWorldApp {
 public static void main(String[] args) {
 if(args.length < 1)
 System.out.println("Hello World!");
 else
 System.out.println("Hello " + args[0]);
 }
}

```

- args.length      array objects have a read-only publicly accessible data member length that specifies number of elements
- "Hello " + args[0] is example of string concatenation

```

class HelloWorldApp {
 public static void main(String[] args) {
 if(args.length<1)
 System.out.println("Hello World!");
 else {
 int num = 0;
 try {
 num = Integer.parseInt(args[0]);
 }
 catch (NumberFormatException nnn) {
 System.out.println("not a number");
 }
 for(int i=0;i<num;i++)
 System.out.println(i + " Hello World");
 }
 }
}

```

*This version expects an integer command line argument, --- the number of times you wish to be greeted.*

## HelloWorldApp.java

- If you have a String that you expect to contain a sequence of digits, you can get the integer value.
- int Integer.parseInt(String)
  - static member function of class Integer
  - note Java naming conventions regarding capitalization

## HelloWorldApp.java

- Suppose the command line argument isn't a number?

```
try {
 num = Integer.parseInt(args[0]);
}
catch (NumberFormatException nnn) {
 System.out.println("not a number");
}
```

- Anywhere there might be an error - throw an exception if one occurs
- If an exception may get thrown, catch it.

## HelloWorldApp.java

- `int num = 0;`      Why not just `int num;?`

## Java programming

Similarities and differences from your C++ experience

## Similarities

- All the basics are essentially identical
  - Expressions used to calculated things
  - Flow control within functions
  - Manipulation of arithmetic data
- Other aspects are closely similar, though not quite identical
  - Variables that are class instances
  - Arguments passed to functions
- *You already know how to program in Java!*

## Differences

- Everything
  - Every function
  - Every data element (apart from local variables of function!)
- Everything **MUST** be declared as a member of a class!
- If something can go wrong, an exception will be thrown; your code **MUST** handle the exception.
- **BUILD** from existing classes, **DO NOT** write everything yourself!

## Java programming – the libraries

- Different approach
  - Don't start with `int main(int argc, char** argv)` and continue on your own, coding functions and maybe creating your own classes [C++ style]
  - Start by designing your program to use instances of standard classes [Java style]

## Java 1.0

- There were 8 packages,  
I never counted the classes

## Java 1.4.2

- There were 135 “packages” (distinct Java libraries) with a total of 2,724 classes.
  - Each class having member functions and data members

... and now in 1.5 ...

## Java 1.5

- There are 166 “packages” (distinct Java libraries) with a total of 3,280 classes.
- These classes are standard part of Java – not application specific classes; they are general purpose utility classes from which you build applications.
- (Now I don’t expect you to know everything about all of them for the exam, just most things about most of them ☺)

## Class?

- Owns
  - What data does an instance own?
  - Are there any data shared by all instances?
- Does
  - What functions does the class provide that let a program exploit the data in a class instance (object).

## Example class – Bank Account

- Owns
  - Identifying number
  - Bank branch number
  - Reference to customer data object of owner
  - Balance
- Does
  - getBalance
  - withdraw
  - deposit

## Example shared data for class

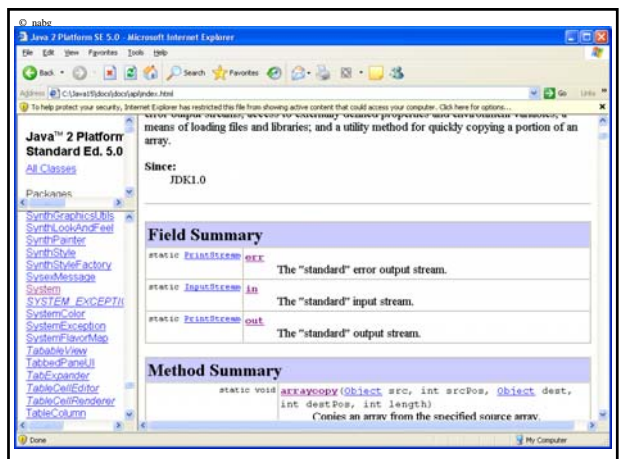
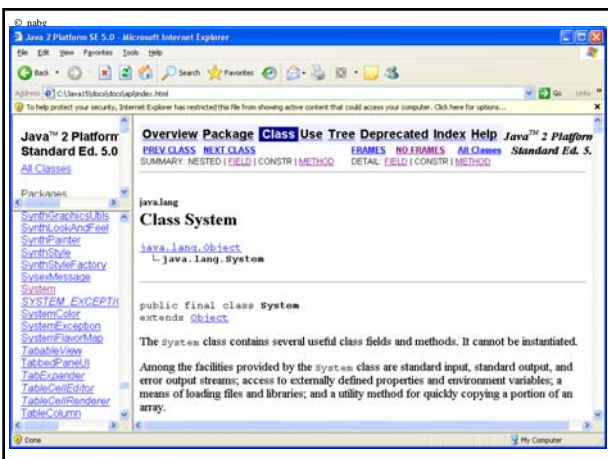
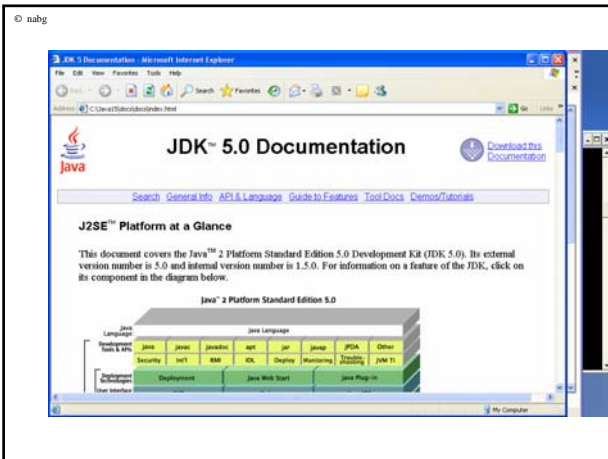
- BankAccount
  - static data elements
    - Interest rate
    - (All accounts get paid interest at the same rate, so only one variable to define this for the class)
  - static functions
    - getInterestRate
    - setInterestRate

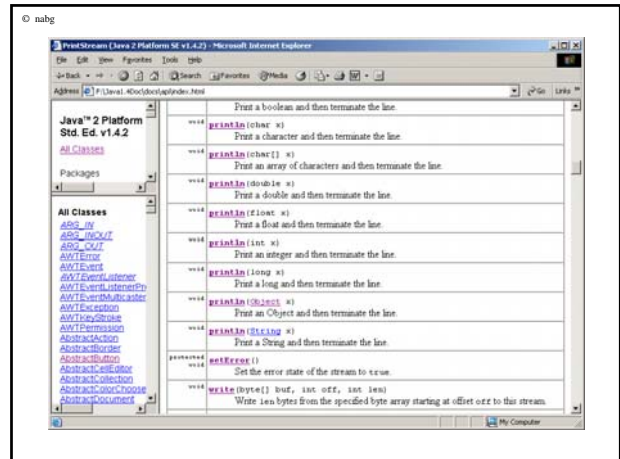
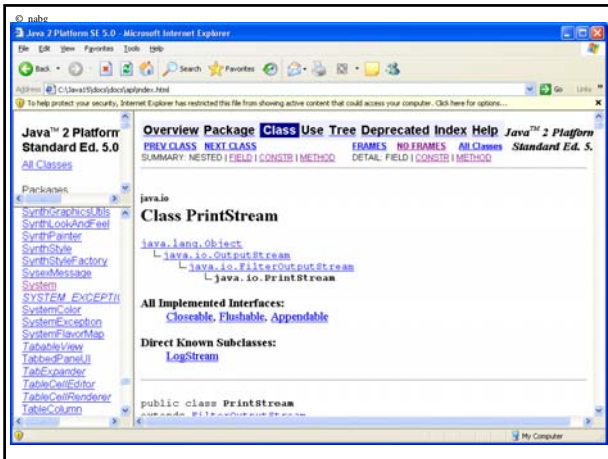
## Class = owns+does

- You think about classes in terms of
  - Owns
  - Does
- So what about the 3000 classes in the standard packages – what do they own, what do they do?

## Ask the class browser!

- A proper OO environment with class libraries also includes a “class browser” that lets you explore the classes
- Traditionally, class browser was special application
- Sun chose to use HTML browser for Java and provided all necessary data in HTML files.





© nabg

## Java & its packages

- You have to develop the confidence to exploit these packages – finding the classes that already exist with the code needed for your application.

© nabg

## Summarizing

- Basic coding constructs are almost identical to those in C++
  - Same flow control
  - Same expressions
- New rule
  - Everything must be a member of a class
- New rule
  - Match class name & name of file containing that class
- New rule
  - Compile “javac XXXX.java”
  - Interpret “java XXXX”
- New rule
  - Use the class browser!