Java
simple programs

Simple programs

- Simply
  - a context to illustrate a few more Java constructs
  - a chance for you to get started fairly quickly in first assignment
- Style
  - "hybrid"; like the C++ you have written
    - Procedural flow
    - Use of instances of a few standard classes

"Hybrid" code!

- I use the term "hybrid" to describe such code, as it is a mix of procedural and class based (term hybrid isn't generally used)
- Thinking about problem is predominantly procedural
  - To accomplish task
    - Do this to data
    - Next do this to data
    - Next do this
    - Finally do this

Calls to subordinate functions that work with global data

Hybrid code – functions and global data structures

- In C++,
  - main() calls other functions defined in same file or in other linked files
  - Data structures defined as global variables (or as filescope variables)
- In Java,
  - Some changes because of requirement that everything be a member of a class
    - Have class that defines main() and the other functions and has class data instead of global/filescope data

Hybrid code – use of classes

- In C++
  - From first program, you used cin and cout – instances of istream and ostream classes – even though you had no idea what a class was
  - Later you added C++'s string class and possibly vector from standard template library
- In Java
  - You've already seen System, PrintStream (System.out = cout) and String

Example 1

- First example
  - to introduce a little more Java I/O
- Program an input file has positive numbers one per line, with 0.0 as sentinel terminator value
  - want mean and standard deviation of these values

Program structure

- Main function
  - gets filename from command line
  - Invokes auxiliary function to do processing.
- Auxiliary function
  - Opens file; terminate if file problems
  - Initializes counters etc;
  - Loops reading and processing data to end of file
  - Computes final results
  - Prints results

Implementations

- Java
  - "Import libraries" that the program depends on
  - Define a class!
    - Can't have "free functions"; everything must belong to a class
    - Class defines a main and a (static) auxiliary function

Code …

- Actual code of functions
  - Iterative and selection constructs are essentially identical to those you learnt in C++
- Everything you have learnt about basic coding in C++ carries over to Java.

Java – overall structure

```
import java.io.*;
public class Example
{
  private static void process(String filename)
  {
  …
  }
  public static void main(String[] args)
  {
     …
  }
}
```

import libraries ~ include headers

This is procedural style, so static main invoking static auxiliary function.
Both functions defined as members of same "class"

Java class

```
public class ClassName
{
  // Members defined in any order
  // Access qualifier (public, protected,
  // private) should be specified for each member
  // (if omitted, get default "package" access –
  // this is a bit sloppy as a coding style)

  // Data members (variables and constants)
  // Member functions (Java prefers term "methods")
  // Members (data and functions) can be
  //  instance members or class (static) members
  // Meaning of "instance" and "class" exactly
  // same as in C++
}
```


Java
Input & Output

- I/O:
  - console (C++'s cin, cout, C's stdin, stdout)
  - local files
  - data entered and displayed using GUI components
  - files accessed across Internet

Input & Output

- Java not well suited to processing lots of data
  - input exceedingly clumsy¶
  - output lacks good formatting capabilities for reports
    - Java 1.5 added C style printf formatted output because so many programmers complained!
- But, sometimes necessary

¶Horstmann got so frustrated with input that he wrote the "console" class described in his book Core Java
Not the only one – many of other introductory text books exploit I/O helper classes written by textbook authors

I/O deficiencies

- Why does Java I/O seem so bad?
  1. C++'s iostream library is actually very sophisticated; you have been spoilt; all those overloaded >> extraction and << insertion operator functions do immense amounts of work for you.
  2. Original concept for Java was mainly for interactive work
     1. Input a single item of data from text-field in a GUI display.
     2. Output data in graphical form, no tabular listings going to lineprinters
  3. (Java development team included programmers who had worked on a complex "streams & filters" model than Sun was pushing for I/O in the 1980s)

Input & Output

- Java I/O model inspired by other work at Sun on "streams" and "filters"
  - you build up something like an input stream by
    - taking a basic connection object (to file, network, ...)
    - giving this to another object that adds capabilities like buffering (for more efficient i/o)
    - giving this buffering object to something that provides a bigger range of input processing functions
    - ...

Input

- Basic input connection – to file, keyboard, network, … -
  - Lets you get one byte at a time
- Buffered connection (text stream)
  - Lets you read one line at a time
    - Result is a String object
  - Bit like a gets() in C/C++'s stdio library or cin.getline() in iostream library

Input

- But typically you don't want to read input as strings representing whole lines, you want to do something like read an integer, a double, and a string all on a line separated by whitespace characters
- C++

```
int ival; double dval; char strval[128];
…
cin >> ival >> dval >> strval;
```


Input – parsing etc

- C++'s istream class's >> operator functions "parse" the data
  - Find the space delimited data elements
  - Interpret numeric input to get a numeric value
- With Java
  - DO IT YOURSELF!
  - (with aid of a few standard functions)

Reading numbers in Java

- One of Java input stream classes provides functions to read and write numbers in internal format ("binary files")
- Other Java classes provide parsing functions that can extract a numeric value from a String (illustrated earlier in HelloWorldApp variant that read a numeric command line argument)
  - suited to text files
  - suited also to GUI inputs that return Strings

Reading numbers

- Normal approach
  - get a String (from file or maybe from some GUI component)
  - attempt to parse string to get numeric value
  - so expect to see something line

    String s = input.readLine()

    numeric n = Numeric class convert String to value

Reading numbers

- If file has several numbers (or data items) per line, it gets a bit more complex
  - have to split string into separate parts, each of which holds one number
  - There are now (at least) two ways of splitting up lines
    - StringTokenizer (you may have used the strtok function in C++ - works remarkably like a StringTokenizer object)
    - split() method in String class

Remember exception handling

- All input operations can fail.
- So all input operations will need to be bracketed in try { } catch ( ) { } exception handling code.

Catch the exception!
But then what do you do?
In most simple programs, all you can do is terminate with an error message.
try { ... } catch { ... } tends to assume that working interactively and have an intelligent user who, when prompted, can re-enter correct data.

Java classes used in Example 1

- File
  - encapsulates concept of file: name, pathname, size, modification date, accessibility, ..
- FileInputStream
  - read individual bytes etc
- InputStreamReader
  - "from byte stream to character stream" (all that Unicode stuff)

This code shows how to build up the streams step by step. Some classes, e.g. FileReader, have been added that have constructors that perform some of the steps shown here in full.

Java classes used in Example 1

- BufferedReader
  - adds buffering, provides a readLine() function that returns a String representing next line of file
- System
  - access to "stdin, stdout", environment variables etc

Java classes used in Example 1

- String
  - a non-editable string
  - (not like string class that you used in C++)
- *Assorted exception classes ...*


```
import java.io.*;

public class Example {

   public static void main(String args[] )
   {
       if(args.length<1)
              System.out.println("Need name of data file");
       else process(args[0]);
   }

    private static void process(String filename)
   {
       ...
   }

}
```

Note difference from C/C++ in numbering of command line arguments

Note:
access qualifiers etc are specified for each individual function and data member
(unlike C++ where can have entries grouped)

Convention for code fragments

- I thought it obvious
- Some students clearly didn't understand
- That slide showed outline with class declaration, and main()
- Body of process() function indicated by ellipsis (…)
- Code shown on some other slide


```
static void process(String filename)
{
  // Open file and package with adapter
  // classes until have BufferedReader from which
  // can read lines

  // loop reading data values (have to handle
  //   exceptions)
  //       read line
  //       attempt to extract double
  //   accumulate sums etc

   // compute mean
   // compute standard deviation
}
```

Code would be "better" as three functions – "setupInput", "readData", and "generateReport". If you can point to a few lines of code and given them an explanatory name, then make that code a separate function


```
static void process(String filename)
{ File      f = new File(filename);
  if(!f.canRead()) {
      System.out.println("Can't read from that file");
      System.exit(1);
      }
  FileInputStream fStream = null;
  try {
      fStream = new FileInputStream(f);
  }
  catch (FileNotFoundException e) {
      System.out.println("No such file?");
      System.exit(1);
      }
  BufferedReader datareader = new
      BufferedReader(new InputStreamReader(fStream));
```


- Using newer Java classes:

```
static void process(String filename)
{     BufferedReader datareader = null;
      try {
            datareader =
            new  BufferedReader(
                  new FileReader(filename));
      }
      catch(FileNotFoundException fne) {
            System.out.println("Couldn't open " +
                  filename);
            System.exit(1);
            }
```

What the … ?

```
File f = new File(filename);
…
FileInputStream fStream = new FileInputStream(f);

BufferedReader datareader =
  new BufferedReader(new
     InputStreamReader(fStream));
```


In the heap (free store)

FileInputStream object
- owns
  - connection to file
- does
  - read one byte

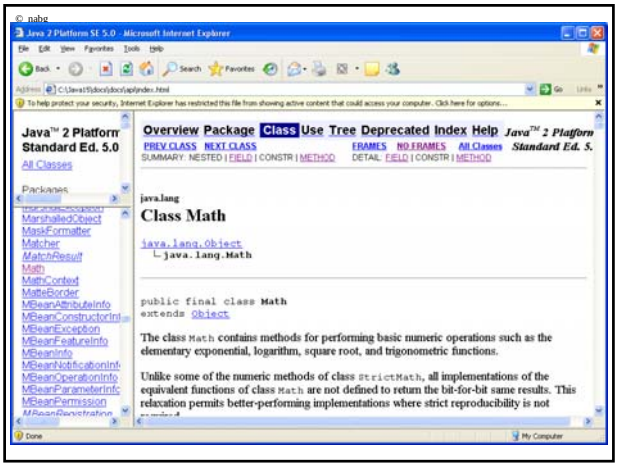
InputStreamReader object
- owns
  - reference to FileInputStream object
  - info on character set in use
- does
  - read one character

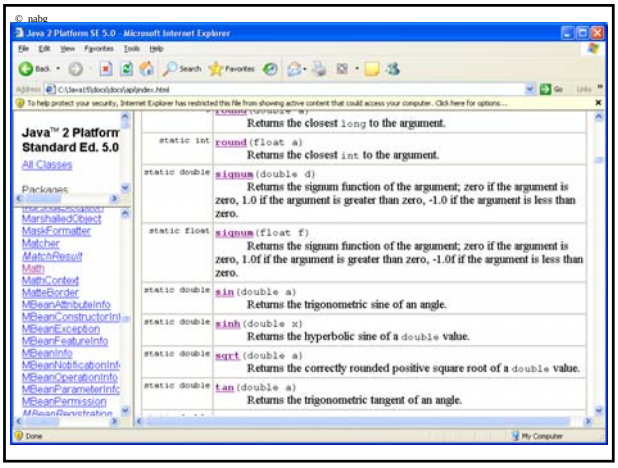
BufferedReader object
- owns
  - reference to InputStreamReader object
  - some space to buffer a line
- does
  - read one line


```
static void Process(String filename)
{
  BufferedReader datareader = …
  ...  // Ellipsis again, that code shown on earlier slide
  int count = 0;
  double sum = 0.0;
  double sumsq = 0.0;
  for(;;) { .. }
  if(count == 0) {
      System.out.println("No data"); return; }
  double mean = sum / count;
  System.out.println("Mean " + mean);
  if(count<2) { System.out.println(
  "Too few items to calculate standard deviation");return;}
  double stdev = Math.sqrt((sumsq - sum*sum/count)
                        /(count-1));
  System.out.println("Standard deviation " + stdev);
}
```


Math

- sqrt function needed
- C/C++?
  - Oh somewhere, sqrt() defined in some library
- Java
  - No such sloppiness!
  - Everything must belong to a class.
  - sqrt() (and cos(), sin() etc) belong to Math class
  - Never have Math objects, Math class simply a "namespace" where maths functions exist





```
for(;;) {
  String s = null;
  try {     s = datareader.readLine();    }
  catch (IOException e) {
      System.out.println("Error reading file");
      System.exit(1);
      }
  if(s==null) break;
  double d = 0.0;
  try { d = Double.parseDouble(s.trim()); }
  catch (NumberFormatException n) {
      System.out.println("Bad data in file");
      System.exit(1);
      }
  if(d == 0.0) break;
  count++; sum += d; sumsq += d*d;
}
```


Example 1 input code

```
try {     s = datareader.readLine();    }
catch (IOException e) {
    System.out.println("Error reading file");
    System.exit(1);
    }
if(s==null) break;
```

- handle case of failure to read a line (e.g. the file didn't have a 0.0 sentinel value as terminator)
- Note, a new String object created each time go round for-loop; then forgotten (relying on automatic garbage collector to remove it)
- *Get a null string at end of file*

Example 1 reading the double

- `d = Double.parseDouble(s.trim());`
- **static double Double.parseDouble(String)**
- Classes Double, Float, Integer define a few useful functions relating to numeric types including static conversion functions that take a string and parse it to yield numeric value (or exception)
- C/C++ lovers will remember atoi() and similar functions


```
/* validate args */
if(!...) { …; return; }
/* input & checks */
…
if(!...) { …; return; }
for(;;) {
        …
        if(…) { …; return; }
        …
        if(…) break;
        …
        if(…) continue;
        …
}
…
```

Aside comment on coding styles

"filter" style

Ooh naughty – multiple exits from loop, multiple returns from a function


```
if(…) {
  …
  if(…) {
        bool littlegreenmansaysoktocross = true;
        bool timetogo = false;
        while(littlegreenmansaysoktocross) {
                …
                if(…) {
                        timetogo = true;
                        littlegreeenmansaysoktocross = false;
                        }
                        else {
                                …
                                if(…) { littlegreenmaysaysoktocross = false; }
                                        else {
                                                ….
                                                if(!...) {
                                                        …
                                                }
                                        }
                                }
        }
        if(!timetogo) {
                …
        }
  }
}
```

"Pascal" style

Java & C++

- Data file 12500 numbers
  - C++ 0.6 seconds
  - Java 3.1 seconds
- Java doesn't compete on performance
- Java competes on convenience; Java's standardized libraries make it much easier to build interactive programs

This example is a little unfair on C++; much of the work is I/O; if had a more intensive computational task then performance difference would be more marked.

Example 2

- Another example
  - arrays
- Program
  - an input file has a set of student records (student name, mark); file starts with count then has string name (just one name, no initials etc), double mark on each line
  - print names of those students who got less than half the average mark

Data file

88
Tom 34
Dragan 48
Harry 71
Sue 68
…
Truls 45
Joseph 13

Purpose of example …

- Illustrates
  - Arrays of numeric values and of Strings
  - Approaches to parsing more complex data inputs

Example 2

- Program structure
  - create arrays
  - loop reading from file, filling in arrays and keeping track of sum of marks
  - calculate average
  - loop through array
    - if mark less than half of average, print name

Example 2

- Java classes mostly as in Example 1
- Java class StringTokenizer from java.util library (package)
  - StringTokenizer is given a String and details of characters that separate ‘tokens’
    - here ‘tokens’ will be character strings representing numbers
    - space will be only separator character
- StringTokenizer now somewhat outdated – also illustrate newer coding style using split() method of class String

Arrays

- An array is essentially a kind of object
  - gets allocated on heap
  - supports [ ] access operator
  - provides “length” accessor
- A variable that is to become an array is declared as an array of specified type
- Space for array allocated by **new** operator (when specify actual array size)

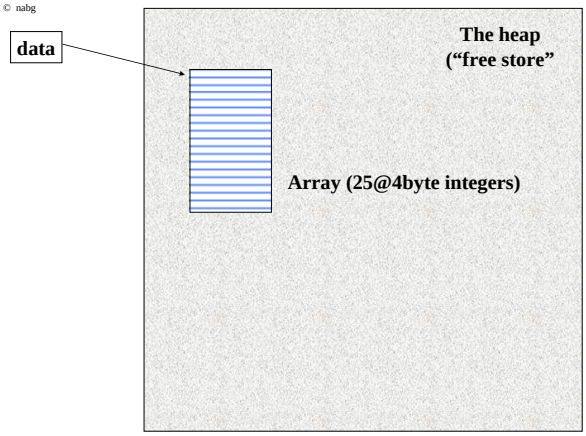
Arrays

- Array of built-in (primitive) types

```
int[] data; // or int data[];
...
data = new int[25];
```

- "data" is an array of integers

```
...
data[3] = 999;
```

Arrays

- Arrays of built in types can also be created with an initializer:

```
int gradepts[] = { 0, 45, 50, 65, 75, 85 };
```


Arrays

- Array of a class type:

```
String[] names;
...
classList = new String[25];
```

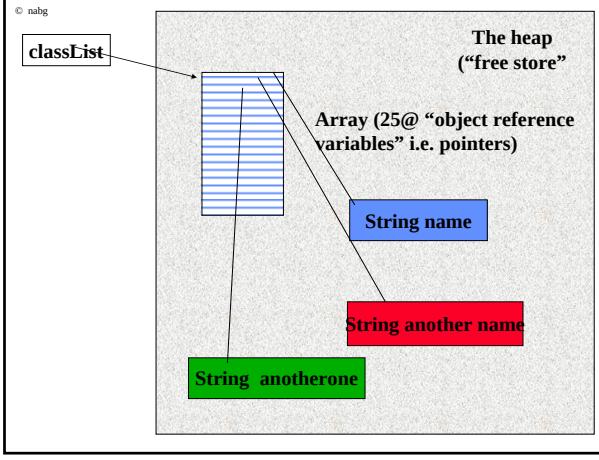
- Now have an array of "object reference variables" (pointers), ***NOT*** an array of Student objects

Arrays

```
classList = new String[25];
```

- array of "object reference variables"
- Create the actual objects separately:

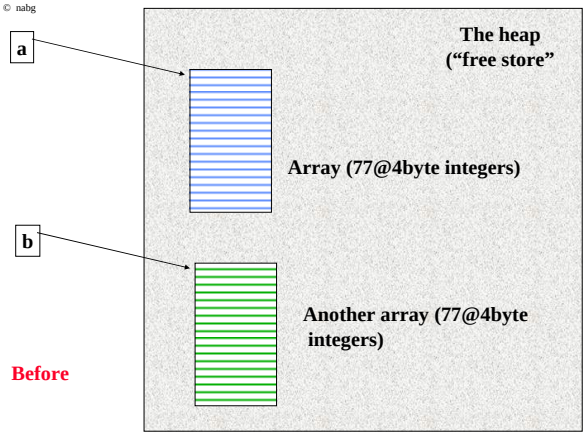
```
for(int i=0;i<25;i++) {
  …
  read in name … create new String
  classList[i] = thename;
```

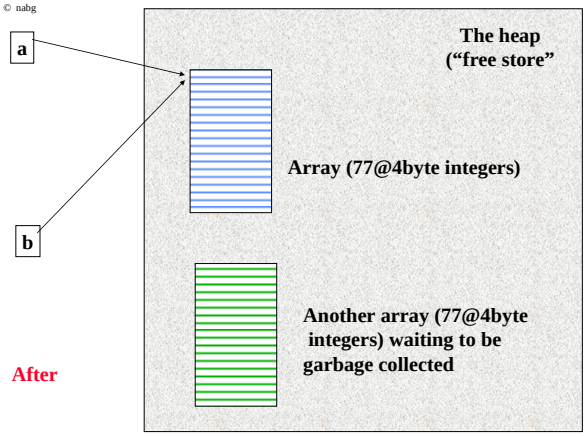



Arrays

- Array variable itself simply a "pointer"

```
int a[];
int b[];
...
a = new int[77];
...
b = new int[77];
...
// This b = a; does not copy contents of array!
b = a;    // Now have two pointers to the same
          // structure, the array that b used
          // to point at will be garbage collected
```



Back to program …

- Processing now sufficiently elaborate that should break it down into more functions.
- These are private auxiliary functions – only called by main()
- Data could be passed from function to function but easier if the functions operate on quasi global data

Procedural code – everything static

- Not defining new type of object
- Simply have set of functions that process data.
- Must define a class, that is Java rule
- All functions must be members of class; again the Java rule
- Functions and data are static


```
imports …
public class Program2 {
  static data members
  private auxiliary functions
  public static void main(String[] args)
  {
    ...
  }

}
```

Program2.java file

```
import java.io.*;
import java.util.StringTokenizer;
public class Program2 {
...
}
```

- Illustrates two slightly different forms of import statement
  - only want one class (StringTokenizer) from util "library" (package), so ask simply for this one
  - want several classes from java.io package; just specify all.
  - Some development environments will put the import statements in for you; they always use the explicit form that names the class; if import several classes from same package then get many import statements

Data needed

- "Global" data – (global scope or filescope in C/C++, static data members of Program class for procedural-style Java)
  - Array of String variables for names
  - Array of int variables for marks
  - int for average mark

Functions

- Function should fit on one screen of your editor,
  if it is longer, break it down into auxiliary functions
- Function call overhead is small
- Little functions, each with a clearly defined purpose, make code more readable

Functions

- main
  - Opens input stream, calls other functions
- getClassSize
  - Reads first line of input, converts to integer, creates arrays
- readData
  - Reads in data with names and marks
- calculateAverage
- printReport


```
public class Program2 {
  private static String[]classList;
  private static int[] marks;
  private static int average;
  public static void main(String args[] )
  {  ...  }
  private static void
     getClassSize(BufferedReader input)
  {  ...  }
  private static void
     readData(BufferedReader input)
  {  ...  }
  private static void calculateAverage()
  {  ...  }
  private static void printReport()
  { … }
}
```



```
public static void main(String args[] )
{
  if(args.length<1) {
      System.out.println("Need name of file with data");
      System.exit(1);
  }
  String fileName = args[0];
  BufferedReader input = null;
  try {
      input = new BufferedReader(
            new FileReader(fileName));
  }
  catch(IOException ioe) {
      System.out.println("Unable to open input file");
      System.exit(1);
  }
      …
}
```

```
public static void main(String args[] )
{
  …
  getClassSize(input);
  readData(input);
  try { input.close(); }
  catch(IOException ioe2) { /* ignore */ }
  calculateAverage();
  printReport();
}
```


main()

- Standard code to create BufferedReader for file
  - Create in main, pass to other functions
  - Don't keep creating new readers (seems popular error by students)
- System.exit(1) – convention for programs that have failed (old Unix convention)
- Handle exceptions
  - Mostly print error message and stop, sometimes can simply ignore
- Invoke other static functions


```
private static void
  getClassSize(BufferedReader input)
{
  int size = 0;
  try {
      String line = input.readLine();
      size = Integer.parseInt(line.trim());
  }
  catch(IOException ioe) {
      System.out.println("Couldn't read from file");
      System.exit(1);
  }
  catch(NumberFormatException nfe) {
      System.out.println("Non-numeric data for class size");
      System.exit(1);
  }
  …
}
```

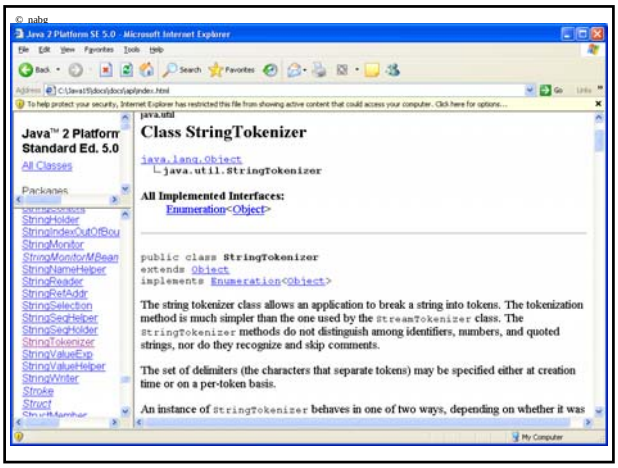


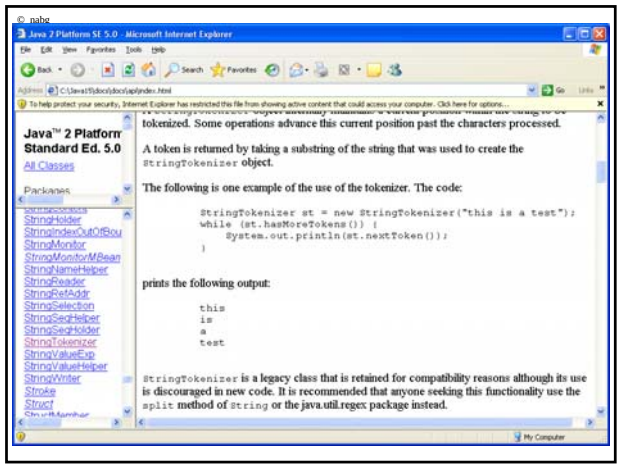
```
private static void
  getClassSize(BufferedReader input)
{
  …
  if(size<=0) {
      System.out.println("Invalid class size");
      System.exit(1);
  }
  classList = new String[size];
  marks = new int[size];
  // System.out.println("Arrays allocated");
  }
```


getClassSize

- Define local variables in smallest scope possible
  - String line only used within try…catch so define it there
- Use tracer statements when developing code
  - System.out.println("some message")
  - At entry and exit from each function
  - Comment out when confident function works
  - Remove such dead code before submitting work for assessment


```
private static void  readData(
  BufferedReader input)
{
  for(int i=0;i<marks.length;i++)
  {
      try {
            String line = input.readLine();
            StringTokenizer strtok = new
                  StringTokenizer(line);
            classList[i] = strtok.nextToken();
            marks[i] =
            Integer.parseInt(strtok.nextToken());
      }
      catch(Exception e) {
            System.out.println("Got exception " +
                  e.toString());
            System.out.println("Occurred on line " +
                  i + " of student data");
            System.exit(1);
      }
  }
  // System.out.println("Data read");
}
```





Combining exceptions with same handler code

```
catch(Exception e) {
  System.out.println("Got exception " +
     e.toString());
  System.out.println("Occurred on line "
     + i + " of student data");
  }
```



```
private static void calculateAverage()
{
  // System.out.println("Calculating average");
  for(int mark : marks) average += mark;
  average = (int) (average/marks.length);
}
private static void printReport()
{
  int cutoff = (int) (average / 2);
  System.out.println(
      "Students with marks less than half of average");
  for(int i=0;i<classList.length;i++)
      if(marks[i] < cutoff)
            System.out.println(classList[i]);
  System.out.println(
      "(They probably plan to pay again next year.)");
}
```


What style of for-loop?

- "for each"
  - When processing each element of collection
- for(int i=0;…;i++) counting loop
  - Indexing into different arrays
  - Need to use the index value
  - …

getClassSize

- Common error
  `String[] classList = new String[size];`
  `int[] marks = new int[size];`
- This declares local variables with same names as the static class member variables.
- Arrays assigned to local variables.
- Local variables destroyed on exit from function, arrays are garbage collected.
- Real arrays were never initialized.
- "Null pointer exception" when try to use arrays in the readData() function

Hint on Assignment 1

- These examples should give you some idea as to how to organize code for assignment 1.

Example 3

- Same as example 2, except now want output with data on pupils sorted by mark.
  - Extra sort and list function added before code to compute average and code to list poorly performing pupils


```
public static void main(String args[] )
{
  …
  getClassSize(input);
  readData(input);
  try { input.close(); }
  catch(IOException ioe2) { /* ignore */ }
  sortAndPrint();
  calculateAverage();
  printReport();
}
```



```
private static void sortAndPrint()
{ // Selection sort – not very efficient
  int count = classList.length;
  for(int i=0;i<count-1;i++){
      int max =i;
      for(int j=i+1;j<count;j++)
            if(marks[j]>marks[max])max=j;
      int itemp = marks[max];
      marks[max]=marks[i];
      marks[i]=itemp;
      String stmp=classList[max];
      classList[max]=classList[i];
      classList[i]=stmp;
  }
  for(int i=0;i<count;i++)
      System.out.println((i+1) + "\t"
            + classList[i] + "\t"
                  + marks[i]);
}
```


Parallel arrays

- When related data stored in separate structures
  - String[] classList (names)
  - int[] marks

  have to swap all elements when sorting otherwise marks end up with wrong pupils!

Data aggregates

Data aggregation

- Why did you have `struct` types and `struct` variables in C/C++?
  - to group together data that belong together

```
struct pupil {
  char* name;
  int mark;
};
```


Java "struct"

- Java doesn't have a distinct `struct` construct but you can achieve much the same thing with a very limited class declaration:

```
public class Pupil
{
  public String name;
  public int mark;
}
```


Java struct-like class

- In Java, such things are almost always promoted into proper classes by adding a few useful methods:

```
public class Pupil
{
   public String name;
   public int mark;
   public Pupil(String aName, int aMark)
   { name = aName; mark = aMark; }
   public String toString()
   {
       return name + " \t " + Integer.toString(mark);
   }
}
```


Minimal methods …

- A constructor
  - Like in C++
    - No return type
    - Name is same as class name
    - May have different versions with different kinds of initializing argument
- A toString() method
  - Signature must be public String toString()

Keep data private

- For various reasons, it is better to keep the data members private
- Since users of class will need to read values of data members, and may need to modify values, *accessor* and *mutator* functions are supplied.
- (A class that provides such functions for all its data members, and which follows certain naming conventions, is called a "bean")


```
public class Pupil
{
   private String name;
   private int mark;
   public Pupil(String aName, int aMark)
   { name = aName; mark = aMark; }
   public String toString()
   {
       return name + " \t " + Integer.toString(mark);
   }
   public String getName() { return name; }
   public int getMark() { return mark; }
   public void setName(String newName)
   { name = newName; }
   public void setMark(int newMark)
   { mark = newMark; }
}
```

Program 4

Yet another version of those students and their marks!

One .java file or several .java files?

- This next version of the program will use the Pupil class.
- Can keep each Java class in own file
  - public class Program in file Program4.java
  - public class Pupil in file Pupil.java
- Or can have just the one file with several classes
  - File Program4.java contains
    - public class Program4
    - class Pupil

Code here illustrates Pupil and Program4 in same file; if doing in NetBeans would use two files


```
import java.io.*;
import java.util.StringTokenizer;

class Pupil
{
  …
}
public class Program4 {
  …
}
```


Program4

1. Use array of Pupil records instead of separate parallel arrays of marks (int[]) and names (String[])
2. File no longer starts with count of records (very inconvenient in practice). Instead, simply read lines until get end of file (a null value returned from readLine()).
3. As don't know size of array, will have to pre-allocate an array we hope will be large enough!
4. Keep count of records read (array length doesn't determine number of actual records).
5. Sort Pupil records.


```
public class Program4 {
  private static Pupil[]     pupils;
  private static int average;
  private static int count;
  private static final int kSIZE=250;
  public static void main(String args[] )
  {
     …
  }
  private static void
      readData(BufferedReader input) {    … }
  private static void calculateAverage() { … }
  private static void printReport() { … }
  private static void sortAndPrint(){ … }
}
```



```
public static void main(String args[] )
{
   if(args.length<1)
        System.out.println("Need name of file with data");
        System.exit(1);
   }
   String fileName = args[0];
   BufferedReader input = null;
   try {
        input = new BufferedReader(
                new FileReader(fileName));
   }
   catch(IOException ioe) {
        System.out.println("Unable to open input file");
        System.exit(1);
   }
   pupils = new Pupil[kSIZE];
   readData(input);
   try { input.close(); }
   catch(IOException ioe2) { /* ignore */ }
   sortAndPrint();
   calculateAverage();
   printReport();
}
```

```
private static void
   readData(BufferedReader input)
{
   count = 0;
   for(;;)
   {
       try {
              String line = input.readLine();
              if(line==null) break;
              StringTokenizer strtok = new
                     StringTokenizer(line);
              String name = strtok.nextToken();
              int mark = Integer.parseInt(strtok.nextToken());
              pupils[count++] =
                     new Pupil(name, mark);
       }
       catch(Exception e) {
              System.out.println("Got exception " +
                     e.toString());
              System.out.println("Occurred on line " +
                     count + " of student data");
              System.exit(1);
       }
   }
```


forever

- Loop

```
for(;;) {
   …
   if(someCondition) break;
   …
}
```

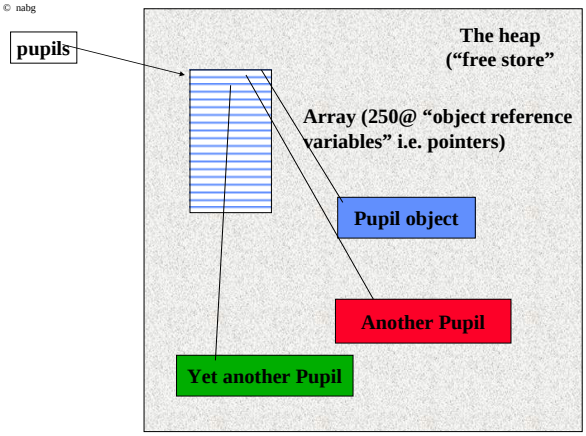
- Very common for handling input streams where detect some form of sentinel end marker (here, end-of-file indicated by null return from readLine

Constructing new object

- Have read name, and mark

```
pupils[count++] =
          new Pupil(name, mark);
```

- Add record to array
  - Array of object type is essentially an array of pointers initialized to null




```
private static void sortAndPrint()
{
  for(int i=0;i<count-1;i++){
     int max =i;
     for(int j=i+1;j<count;j++)
          if(pupils[j].getMark()>
               pupils[max].getMark())max=j;
     Pupil ptemp = pupils[max];
     pupils[max]=pupils[i];
     pupils[i]=ptemp;
  }
  for(int i=0;i<count;i++)
     System.out.println((i+1) + "\t"
               + pupils[i]);
  }
```


Invoking method of object in array

```
if(pupils[j].getMark()>
```

- pupils[j] array element, an object of type Pupil
- .getMark() call member function

Have to use counting loops

- foreach loop
  - for(Pupil p : pupils)

  will process every element of array
- How many elements? kSIZE i.e. 250
- How many elements actually exist?
  - Don't know, it depends on data in file but less than 250
- Would encounter a null pointer exception


```
private static void calculateAverage()
{
  for(int i=0;i<count;i++) {
     average += pupils[i].getMark();
  }
  average = (int) (average/count);
}
private static void printReport()
{
  int cutoff = (int) (average / 2);
  System.out.println("Students with marks less than
  half of average");
  for(int i=0;i<count;i++)
     if(pupils[i].getMark() < cutoff)
          System.out.println(pupils[i]);
}
```


Java is smarter than that …

Java knows how to sort …

You should have expected that …

- C/C++ libraries include a qsort() function that can sort arrays of structures;
  of course, you have to supply an auxiliary function that defines how to determine which struct is greater

```
void  qsort(void  *base,
     size_t  nel,  size_t  width,
     int
     (*compar)(const void *,
          const void *));
```


OK, which one of you can explain

```
void  qsort(void  *base,  size_t  nel,
 size_t  width,
 int
    (*compar)
    (const void *, const void *)
);
```


The qsort() function is an implementation of the quick-sort algorithm. It sorts a table of data in place. The contents of the table are sorted in ascending order according to the user-supplied comparison function.

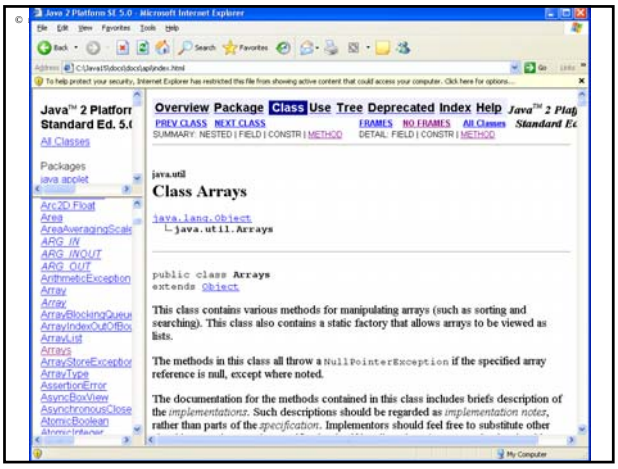
The base argument points to the element at the base of the table. The nel argument is the number of elements in the table. The width argument specifies the size of each element in bytes. The compar argument is the name of the comparison function, which is called with two arguments that point to the elements being compared.

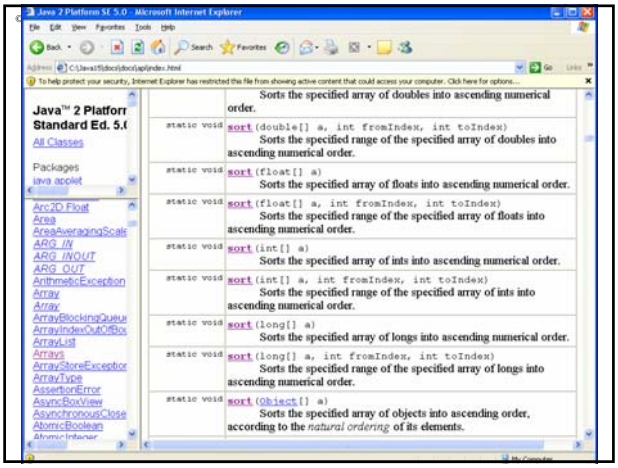
The function must return an integer less than, equal to, or greater than zero to indicate if the first argument is to be considered less than, equal to, or greater than the second argument.

The contents of the table are sorted in ascending order according to the user supplied comparison function.

Java's sort

- Naturally, in Java, any function for sorting must be defined as part of a class;
  it is the class Arrays
- Arrays is not a class from which one creates "Arrays" objects;
  it is simply a namespace thing;
- Arrays is a class where functions for sorting, binary search etc are defined.





Arrays.sort(array, start-index, end-index)

- Arrays.sort(array)
- Arrays.sort(array, start, end)
- Lots of versions for different kinds of arrays
  - int[]
  - float[]
  - Object[] ? !

Arrays.sort

- Obviously Java knows how to compare int, float, double, char, and other built-in types.
- But what about objects that are from classes that Java has never heard about before,
  e.g. class Pupil
- As with C++/C's qsort(), you must supply the function that does the comparison

interface Comparable

- Simplest approach …

```
interface Comparable {
  int compareTo(Object other);
}
interface Comparble<T> {
  int compareTo(T other);
}
```

class X implements Comparable

- You define your class as implementing Comparable interface.
- You define a compareTo function that determines ordering when comparing "this" object with "other" object
  - Return –ve value if "this" is smaller than "other"
  - Return 0 if they are equal
  - Return +ve value if "this" is larger than "other"

See details with the "Boxes" example associated with assignment 1

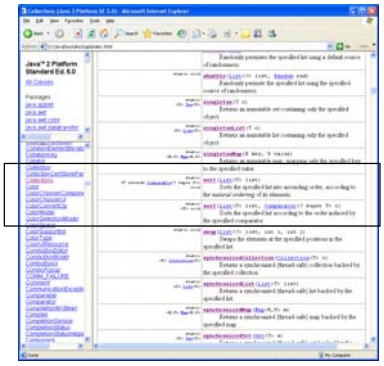
Program 5

- Same as Program 4
- But using Arrays.sort rather than our own.
- Class Arrays is in java.util package, so need another import statement.

NetBeans of course will look after the imports for you, but you should try to understand what is going on

Collections.sort

- Java has collection classes like Vector (a dynamic array), LinkedList and ArrayList.
- May need to sort data held in a collection.
  - Copy from collection into temporary array
  - Use Array.sort()
- To save you the trouble, there is another helper class Collections that is like Array



sort(List<T> list)
Sorts the specified list into ascending order, according to the *natural ordering* of its elements.static
sort(List<T> list, Comparator<? super T> c)
Sorts the specified list according to the order induced by the specified comparator.

How does it work? Well it copies into an array and uses Array.sort


```
class Pupil implements Comparable
{
  …
  public int compareTo(Object other)
  {
      Pupil otherP = (Pupil) other;
  // Don't want normal numeric sort which would be
  // int dif = mark - otherP.mark;
  // as that would put them in ascending order by
  // mark
  // Want descending order by mark
      int dif = otherP.mark - mark;
      if(dif != 0) return dif;
  // If same mark, want them alphabetical
  // sorted by name
      return name.compareTo(otherP.name);
  }

}
```

Java 1.4 and earlier version

Pupil.compareTo(Object other)

- Should only be comparing Pupil objects with other Pupil objects,
  and can only do comparison if can look inside objects and get at their fields
- Hence type cast

```
Pupil otherP = (Pupil) other;
```

- Compare first by mark, if marks are equal compare by name
- Strings already have a compareTo function that sorts alphabetically


```
private static void sortAndPrint()
{
  Arrays.sort(pupils, 0, count);
  for(int i=0;i<count;i++)
     System.out.println((i+1) + "\t"
          + pupils[i]);
}
```



```
class Pupil implements Comparable<Pupil>
{
  …
  public int compareTo(Pupil other)
  {
  // Want descending order by mark
     int dif = other.mark - mark;
     if(dif != 0) return dif;
  // If same mark, want them alphabetical
  // sorted by name
     return name.compareTo(other.name);
  }

}
```

Java 1.5 version (Program6)

Java is smarter than that …

Use collection classes …

Knowing how big the arrays are

- Earlier examples cheated
  - Data file started with a line giving number of records
  - This number was read and used to determine size of array to create.
- More generally, you don't know how many data elements there are to process, you must simply process all in the input file.
- So used scheme where pre-allocated the array.
- But how do you create an array of appropriate size?

Guess a size?

```
public class Program {
  private static final int
       kIMSURETHISWILLBEBIGENOUGH = 10000l;
  private static Pupil[]      enrollment;
  public static void main(String args[] )
  {
       enrollment =
              new Pupil[kIMSURETHISWILLBEBIGENOUGH ];
       if(args.length<1)
              System.out.println("Need name of data file");
       else process(args[0]);
  }
       static void process(String filename)
  {
       ...
  }
}
```

Trouble with guessing …

- Too large
  - Waste of storage space
- Too small
  - Run time exception, array subscript out of bounds

kSIZE=250; well it would work for CSCI subjects now, but before the engineers were moved into CSCI191, the combined CSCI114/CSCI191 class had 300+ students and the program would have failed if applied to data file for that subject

Dynamic arrays

- You might have been shown how to implement a "dynamic array" in your C++ studies
  - Create array with initial size
  - Add elements
    - If array is full when want to add an element
      - Create a new larger array
      - Copy existing data
      - Get rid of too small array
      - Continue working with larger array

Using a dynamic array

- You could code one yourself in Java …
  - A variable to reference the array
  - An integer to represent current size
  - An integer to count number of elements added
  - Initialize – allocate small array, record size, zero count
  - Loop reading data
    - Create new element
    - Invoke an add routine
  - Add routine
    - Grow array if necessary


```
private static Pupil[] enrollment;
private static int size;
private static int count;
public static void main(String[] args)
{
  enrollment = new Pupil[100]; size = 100; count = 0;
  …
}
private static void process(String filename)
{
  …;
  for(;;) {  …; Pupil next = new Pupil(id,mark);
      add(next); … }
}
private static void add(Pupil s)
{
  if(count==size) {
      Pupil[] temp = new Pupil[size+100]
      for(int i=0;i<size;i++) temp[i]=enrollment[i];
      enrollment = temp; size +=100;
  }
  enrollment[count++] = s;
```

Make the array grow

Java has everything built-in!

- You don't need to code such things.
- Java comes with a number of "Collection" classes.
  - Linkedlists,
  - Resizable arrays,
  - Binary tree structures,
  - Hashtables
  - …
- All in java.util package.

Using collection classes

- Pick the java.util class most suited to current needs
  - Simple collections: e.g. Vector, LinkedList, ArrayList, …
  - Ordered collections: Treeset, …
  - Keyed collections: Hashmap, …
- Create an instance of that class
- Add your data to the collection

Complication here …

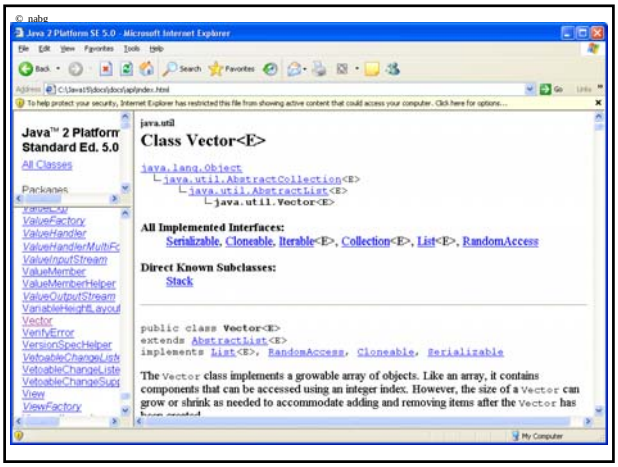
- Traditional Java style
- Java 1.5 style – greater type security

- You need to be familiar with both
  - Very large volume of existing code in traditional style – you have to work with and update this code
  - New style for new applications

- Old style first

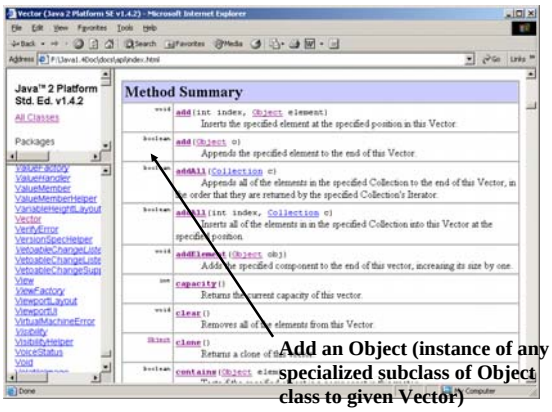
The simplest collection class

- Simplest collection class in java.util:
  - Vector
- java.util.Vector is essentially a class defining a simple dynamic array
  - Constructors
    - No argument, get small array to start with;
    - integer argument – define starting size;
  - add (and addElement) methods – add more data, growing array if necessary
  - elementAt(int ndx) method to get an element
  - size() to get number of elements



Collections of what?

- Collections of Objects! What else did you expect.
- Object - topic of next lecture.
- Java differs from C++ here
  - C++: classes are independent
  - Java: classes are all part of a great hierarchy, every class is inherently a specialized subclass of Java's Object class
  - So, every instance of every class is an Object



Objects in Collection

- Can put any Object into a collection
  - Most of the collection classes have an add(Object o) method

- Can get at elements in a collection; methods vary a little, Vector has methods like
  - firstElement(), lastElement(), elemenAt(int index)
- What do you get when you ask a collection for a particular element using one of the access methods?

Object in => Objects out

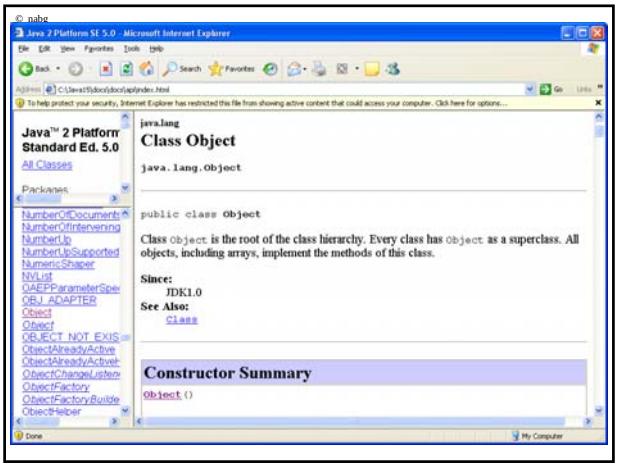
Object **elementAt**(int index)

Returns the component at the specified index

- Get back an Object of course.

What is an "Object"?

- More in next lecture.
- Base class for all Java classes.
- Defines some properties that all Java Objects have
  - Functions that work with every object
    - toString – every object can print out some information, though the default is simply to print its class name and memory address (which isn't usually very informative)
    - Functions that you learn about later for locking access to objects



Type casts

- You know what you put in a collection, e.g. a Pupil record.

So

- You know that the Object you get back isn't just an Object, it really is a Pupil record

So

- You type cast when you get something out of a collection:

Pupil one = (Pupil) myCollection.firstElement();

Java 1.5 enhancement

- Java 1.5 has "template collections"
- Look a bit like C++ templates, work quite differently.
- In Java 1.5
  - When you define a collection object (instance of any collection class) you specify the class of the objects that will be stored in that collection
  - Compile time checks make sure you are using the collection correctly (in some situations, supplemented by run-time checks)
  - When you obtain an object from the collection, you don't have to do an explicit type cast – casting is automatic

1.5

Vector<Pupil>

```
Vector<Pupil>      collection;
…
collection = new Vector<Pupil>();
…
Pupil s = new Pupil(…);
collection.add(s);
..
Pupil x = collection.firstElement(); // No explicit type cast
```

Demo program

- Class Pupil
- Program
  - Main creates a Vector
  - Process
    - Loop reading data, creating Pupil record objects and adding them to Vector
    - When complete, data moved from Vector to Pupil[]

Java 1.4 style

```
private static void readData(BufferedReader input)
{
   Vector collection = new Vector();
   count = 0;
   for(;;)
   {
        try {
               String line = input.readLine();
               if(line==null) break;
               StringTokenizer strtok = new
                      StringTokenizer(line);
               String name = strtok.nextToken();
               int mark = Integer.parseInt(strtok.nextToken());
               collection.add(new Pupil(name, mark));
               count++;
        }
        catch(Exception e) {
               …
               System.exit(1);
        }
   }
   pupils = (Pupil[])
        collection.toArray(new Pupil[0]);
}
```


Java 1.5 style

```
private static void readData(BufferedReader input)
{
   Vector<Pupil> collection = new Vector<Pupil>();
   count = 0;
   for(;;)
   {
        try {
               String line = input.readLine();
               if(line==null) break;
               StringTokenizer strtok = new
                      StringTokenizer(line);
               String name = strtok.nextToken();
               int mark = Integer.parseInt(strtok.nextToken());
               collection.add(new Pupil(name, mark));
               count++;
        }
        catch(Exception e) {
               …
               System.exit(1);
        }
   }
   pupils = collection.toArray(new Pupil[0]);
}
```


Use collection or convert to array?

- Depends on what processing you need to do later.
- Usually, just work with collection (Vector, LinkedList, whatever)
- Sometimes more convenient to get data copied from collection to array.
- Most collections have a toArray() method
- Here wanted the array form for next sorting step

Vector.toArray()

```
enrollment = collection.toArray(
    new Pupil[0]);
```

- Why the dummy argument new Pupil[0]?

Java 1.5 “improvement”

StringTokenizer deprecated

Managing without StringTokenizer

- StringTokenizer class is "deprecated"
  - Means that you should not use it in any new programs
- String class got given extra methods in Java 1.4 release
  - Including a split() function

String[] String.split(regex-pattern)

- Split method breaks up a String returning an array of String objects for the separate parts.
- Regex-pattern – a "regular expression" that defines how to recognize the points where string is to be split
- Regular expressions can be quite complex, you may get short introduction in one of C++ subjects.

split() rather than StringTokenizer

- Sun's justification:
  - Regex-es allow for definition of much more sophisticated criteria for splitting a string
  - So can do much more than StringTokenizer
- Consequence
  - Need much more sophisticated programmers to use them correctly!


```
void readData(BufferedReader input)
  {
      …;
      String s = input.readLine();
      String[] sparts = s.split("\\s")
      name = Integer.parseInt(sparts[0]);
      mark = Integer.parseInt(sparts[1]);
      …
  }
```


Aside comment!

Regex-es (yeech)

```
String[] sparts = s.split("\\s")
```

- Why \\s?
  All we wanted to say was split at whitespace characters (which StringTokenizer does by default).

Aside comment!

Regex-es

- A test program-

```
public class Tez {
  public static void main(String[] args)
  {
      String query = "This-is-a-test";
      String[] parts = query.split("-");
      System.out.println(
            "There were " + parts.length + " parts");
      for(int i=0;i<parts.length;i++)
            System.out.println(parts[i]);
  }
}
```

Aside comment!

Regex-es

- Regex can be
  - A simple character, split at "-"
    - query.split("-")
    - Result: ["This", "is", "a", "test"]
  - A set of characters, split at any vowel "[aeiou]"
    - query.split("[aeiou]");
    - Result: ["Th", "s-", "s-", "-t", "st"];
  - Sets are defined "[" characters "]"

Aside comment!

Regex-es

- There are predefined sets represented by escape combinations (see documentation)
  - \d set of all digits
  - \D set of all characters that are not digits
  - \s set of all whitespace characters
  - \S set of all characters other than whitespace
  - Others

Aside comment!

Regex-es

- So, we wanted to say "split at whitespace" so could use the predefined character set \s
- But had to define this as a string!
- In a string, the \ character is an "escape character" (same as it was in C++).
- So had to "escape the escape" - hence "\\s"

Aside comment!

Regex-es

- Other problems
  - Characters like [].|+* have special meaning inside regexes
  - So if want to split at e.g. * character, cannot say
    - String query = "This*is*a*test";
    - query.split("*");
  - (You would get a runtime error about a bad regex)
  - Instead must use an escape combination-
    - query.split("*");

Last aside comment!

Regexes

- Regex-es
  - Fun (?)
  - Powerful
  - More generally used for much fancier tricks than simply splitting up a string
  - Something to learn about in some C++ subject (or in context of Perl, in CSCI399)
  - A pain
- But as Sun expects us to use more sophisticated features added to Java, I guess we just learn to use regexes

Working with collections

Iterating through collections

- Can use a for loop

```
Vector<Pupil> collection = new Vector<Pupil>();
…
// fill collection with data
…
int numelements = collection.size();
for(int i=0;i<numelements;i++) {
  Pupil p = collection.elementAt(i);
  …
}
```


Iterating through collections

- (As explained in CSCI124 (?)), collections normally have associated Iterator objects that can be used when seeking to iterate through collection.
- Use of iterators tends to result in cleaner, more readily understood code

Uhm, revised healthier low-class CSCI124 probably hasn't touched on collection classes and their Iterators.


```
private static void process(String filename)
{
  …
  for(;;) { … }
  Iterator<Pupil> iter =
    collection.iterator();
  while(iter.hasNext()) {
    Pupil st = iter.next();
    double diff = mean - st.fMark;
    diff = Math.abs(diff);
     if(diff > 2.0*stdev)
      System.out.println(st);
  }
}
```


and another Java 1.5 feature …

- Very common to:
  - create and fill a collection
  - get Iterator from collection
  - have a loop with hasNext() and next() operations on Iterator
- Java 1.5 adds a little "syntactic sugar" with its "for each" loop
- Code using the "for each" construct slightly more readible
- Use "for each" loop if really do want to process all elements in turn.
- Use explicit Iterator when want to do something fancier such as remove elements while walking the collection (Iterator class has a remove operation)

Iterators

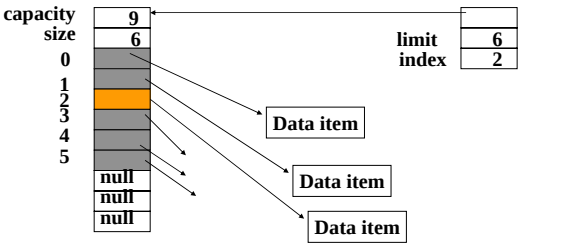
- Most of the java.util collection classes have a method iterator() that returns an Iterator.
- Iterator defines methods
  - hasNext()
  - next()

Iterator

- Different collections yield different iterator objects (instances of specialized Iterator classes)
- They all have the same operations
  - hasNext()
  - next()

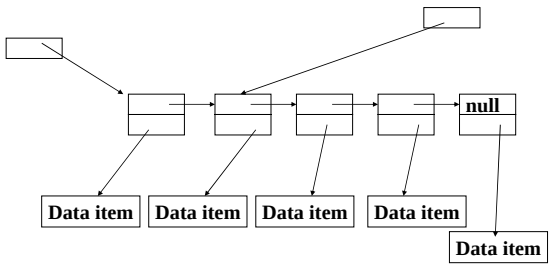
Iterator - vector

- Vector (dynamic array)



Iterator linkedlist

- LinkedList



Iterators

- No reason to know their internal structure
- No reason to know what specialized class they are instances of.
- They all work the same, just give me an Iterator

Enumeration

- Some collection classes give you an Enumeration instead of, or as an alternative to an Iterator.
- Enumeration was Java's first version of "iterator" construct.
- Java developers decided that they didn't like some details of its implementation so they later invented Iterator
- Enumerator remains for backwards compatibility

"for-each" loop/iterator/enumeration

- Which to use?
  - "for-each": use when working through all elements of array or a collection such as Vector or LinkedList
    - For-each will actually use an Iterator
  - Iterator: use when working through a collection that may get modified (Iterator.remove() operation)
  - Enumeration: use only when necessary for compatibility with "legacy" code

End of examples

- OK
  - Not really Java style
  - Haven't really explained Objects and object reference variables yet
  - But enough to get you started on Java coding and assignment 1
    - Actual code – iteration constructs, selection constructs etc – really no difference from C++
    - Structure of program – few extra restrictions, no free functions (everything is part of a class)
    - Minor syntactic differences from C++ (e.g. setting access control on class members)
    - But what you already know from C++ mostly works!

Minor issues

-programming style

-Primitive type data elements in collections

Side issue 1 : "main() must create an object"

Note on stylistic convention followed in some textbooks

- Many textbooks are written by authors who believe that you should ***never*** write any "procedural" style Java
  (they have never read any of Sun's examples of more advanced applications!)
- They use a style where an object is always created, and the code is executed by this object

Version of an earlier example in this alternative style

- Class ProgramWithObject – same program really
  - main() now creates an instance of its own class and invokes a method on that instance
  - All data members (apart from any constants) and function members now become instance members rather than class members.
  - (Code uses default no argument constructor that is provided automatically if you don't define any constructors)


```
public class ProgramWithObject {
   private Pupil[] pupils;
   …
   public static void main(String args[] ) {
        ProgramWithObject iReallyInsistOnHavingAnObject =
              new ProgramWithObject();
        iReallyInsistOnHavingAnObject.work(args);
   }
   private void work(String args[]) {
        System.out.println("This is object " +
               this.toString());
        System.out.println("I am starting work");
        this.pupils = new Pupil[kSIZE];
        if(args.length<1)
              System.out.println("Need name of data file");
        else this.process(args[0]);
   }
    private void process(String filename)
    { … }

}
```


Output from modified program

$ java ProgramWithObject data1
This is object ProgramWithObject@16f0472
I am starting work
…

"this"

- Remember "this" in C++?
  - Pointer to current instance of class, the object for which a method is being executed.
- You probably never used "this", the C++ compiler takes it as implicit when dealing with member functions and data members.
- Rules are similar in Java.
  - Can use "this" to reference the current object for which a method is being executed.
  - Usually don't bother as compiler takes it as implicit.

Side issue 2: collection of numeric values

- Another area where 1.5 differs from earlier Javas.
- Collections are collections of Objects
  - int, float, long, double etc are not objects
  - values have to be "boxed" inside objects before can go in collection
- Old Javas – programmer did the "boxing"
- Java 1.5 – boxing is automatic
- Old style first

Another side issue – collections and primitive types

- java.util collection classes are collections of Objects
- Suppose you wanted a collection of integer values; how about …

```
Vector intdata = new Vector();
for(;;) {
  // read data from file or somewhere
  …
  int nextOne = …;
  intdata.add(nextOne); //???????????? !!!!!!
```

Signature of function was: boolean add(Object o)

Collections and primitive types

- That code would not compile with Java 1.4.
- Collections like Vector are collections of Objects (things that exist as structures in the "heap")
- int (double) values are held in stack-based variables, they don't exist in the heap.
- If you want to put the value of some primitive type variable in a collection, you must "box" the value in an object

Integer, Double, Float, …

- These classes exist for you to "box" primitive values inside heap based objects so that you can use them in contexts where an object is required (which mainly means when you want to put them into collections!).

```
Vector intdata = new Vector();
for(;;) {
  // read data from file or somewhere
  …
  int nextOne = …;
  Integer boxedNextOne = new Integer(nextOne)
  intdata.add(boxedNextOne);
```


Integer, Double, Float

- Objects of these classes are immutable – you can not change their values
- So you cannot do arithmetic on them.
- If you have "boxed" an int value in an Integer object, and need to do arithmetic
  - Extract value from Integer object
  - Do arithmetic
  - Create a new Integer object to hold the result

Updating a value in a collection

```
Vector v = new Vector();
…
for(;;) {
  …
  int val = …;
  Integer bval  = new Integer(val);
  v.add(bval);
  …
}
…
// increment value at 3rd entry in Vector (0 based array)
Integer tmp = (Integer) v.elementAt(2);
int oval = tmp.intValue();
oval += whatever;
Integer replace = new Integer(oval);
v.setElementAt(replace, 2);
```

Java 1.5 – autobox (and unbox)

```
Vector<Integer> collection = new Vector<Integer>();
…
collection.add(37);
…
collection.add(2123);
…

int value = collection.lastElement();
```