

## CMSC433, Spring 2003 Programming Language Technology and Paradigms Basic Java

## Outline

- Object oriented programming principles
  - How Java realizes them
  - How Java differs from C++
- Useful information on Java (not covered in class)
  - Using the compiler
  - I/O libraries
  - Container classes

CMCS 433, Spring 2003

2

## Software Engineering Goals

- Reliability (it works!)
- Performance
- Reusability (write-once, then reuse)
- Maintainability
  - Easy to modify/extend
  - Easy to understand
- Quick development time

CMCS 433, Spring 2003

3

## Object Orientation

- Abstraction
  - focus on essential properties, ignore unimportant details
- Encapsulation
  - separate external, visible behavior from internal, hidden behavior

CMCS 433, Spring 2003

4

## Object Orientation (cont'd)

- Combining data and behavior
  - objects, not developers, decide how to carry out operations
- Sharing
  - similar operations and structures are implemented once
- Emphasis on object-structure rather than procedure structure
  - behavior more stable than implementation
  - ... but procedure structure still useful

CMCS 433, Spring 2003

5

## And one more ...

- Security and Reliability
  - Write code that works, and is not insecure

CMCS 433, Spring 2003

6

## Java

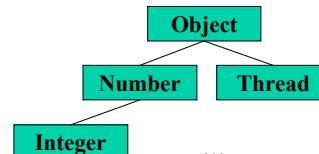
- Similar to C++, but with “unsafe” features removed, and others added
- Fully specified, compiles to virtual machine
  - machine-independent
- Secure
  - bytecode verification (“type-safe”)
  - security manager

CMCS 433, Spring 2003

7

## Java design

- Everything inherits from **Object**\*
  - Allows sharing, generics, and more



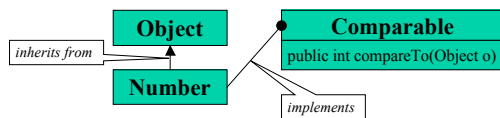
\* Well, almost: there are primitive int, long, float, etc.

CMCS 433, Spring 2003

8

## Java Design

- *Inheritance*
  - Hierarchical code sharing (“is-a”)
- *Interfaces*
  - For “mixins” & non-hierarchical frameworks



CMCS 433, Spring 2003

9

## Java Design

- Security and reliability
  - Strong type system
    - String s = (String) new Integer(42); **not allowed!**
  - Garbage collection
    - No free()
  - Exceptions
    - Separation of error-handling from algorithm

CMCS 433, Spring 2003

10

## Java libraries and features

- Utilities
  - collection classes, Zip files, internationalization
- GUIs, graphics and media
- Networking
  - sockets, URLs, RMI, CORBA
- Threads
- Databases
- Cryptography/security

CMCS 433, Spring 2003

11

## Some of what's missing from C++

- Preprocessor (#include, #define, ...)
- Some “low-level” types
  - structs and unions
  - enumerated types
  - bit-fields
- Some function features
  - variable-length argument lists
  - operator overloading
- Some class features
  - multiple inheritance (of implementation)
  - templates/ parameterized types (but in 1.5, beta available now!)

CMCS 433, Spring 2003

12

## Naming conventions

- Classes/Interfaces start with a capital letter
  - **Object**, **Number**, **Thread**, ...
- packages/methods/variables start lowercase
  - **Thread** myThread = new **Thread**();
  - java.lang, org.xml.sax
- Capitalize multi-word names (no underscores)
  - **SortedList**, **compareTo**, **toBinaryString**
- CONSTANTS all in uppercase (use underscores)
  - **PI**, **E**, **MAX\_VALUE**

CMCS 433, Spring 2003

13

## Object-oriented programming in Java

## Java Classes and Objects

- Each object is an instance of a class
  - an array is an object
- Each class extends *one* superclass
  - **Object** if not specified
  - class **Object** has no superclass

CMCS 433, Spring 2003

15

## Objects have methods

- All objects, therefore, inherit them
  - Default implementations may not be the ones you want

```
public boolean equals (Object that) "conceptual" equality
public String toString() returns print representation
public int hashCode() key for hash table
public void finalize() called when object garbage-collected
```

- And others ...

CMCS 433, Spring 2003

16

## Objects and references

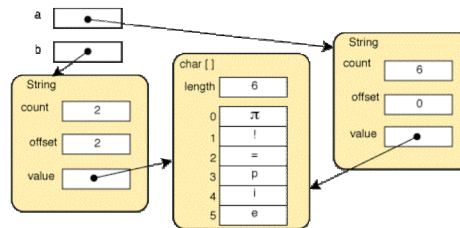
- All objects allocated on the heap with *new()*
  - All variables of non-primitive type are references to an object or *null*; assignment (=) copies references
  - No object can “contain” another object
  - No objects stack-allocated (only references there)
- Reference is like a C++ pointer, except
  - can only point to start of heap-allocated object
  - no pointer arithmetic allowed
  - use *.* instead of *->* to access fields/methods

CMCS 433, Spring 2003

17

## String example

```
String a = "π!-pie";
String b = a.substring(2,4);
```



CMCS 433, Spring 2003

18

## Equality

- **Object** *.equals()* method
  - Structural (“conceptual”) equality
  - reflexive, symmetric, transitive
- *==* operator
  - true if arguments reference *the same object*
  - *o == p* implies *o.equals(p)*

CMCS 433, Spring 2003

19

## class Complex – a toy example

```
public class Complex {
    private double r, i;
    public Complex(double r, double i) {
        this.r = r;
        this.i = i;
    }
    public String toString() {
        return "(" + r + "," + i + ")";
    }
}

public Complex plus(Complex that) {
    return new Complex(
        r + that.r,
        i + that.i);
}
```

CMCS 433, Spring 2003

20

## Using Complex

```
public static void main(String[] args) {
    Complex a = new Complex(5.5,9.2);
    Complex b = new Complex(2.3,-5.1);
    Complex c,d;
    c = a.plus(b);
    d = a.plus(b);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("c.equals(d): " + (c.equals(d)));
    System.out.println("c = d: " + (c==d));
}
```

prints:  
a = (5.5,9.2)  
b = (2.3,-5.1)  
c = (7.8,4.1)  
c.equals(d): false  
c == d: false

CMCS 433, Spring 2003

21

## Adding equals to Complex

```
public class Complex {
    ...
    public boolean equals(Object o) {
        if (this.getClass().equals(o.getClass())) {
            Complex c = (Complex)o;
            return c.r == this.r && c.i == this.i;
        } else {
            return false;
        }
    }
}
```

Runtime test to determine the object's actual class

CMCS 433, Spring 2003

22

## Equal objects have equal hashCodes

```
public class Complex {
    ...
    // This is more complicated than I'd like it to be for this
    // introduction, but code must be right before it can be
    // simple
    public int hashCode() {
        long rHash = Double.doubleToLongBits(r);
        long iHash = Double.doubleToLongBits(i);
        return (int)( rhash ^ (rHash >> 32)
            ^ (iHash >> 16) ^ (iHash >>> 48) ^ (iHash << 8);
    }
}
```

CMCS 433, Spring 2003

23

## Using Complex again

```
public static void main(String[] args) {
    Complex a = new Complex(5.5,9.2);
    Complex b = new Complex(2.3,-5.1);
    Complex c,d;
    c = a.plus(b);
    d = a.plus(b);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("c.equals(d): " + (c.equals(d)));
    System.out.println("c = d: " + (c==d));
}
```

prints:  
a = (5.5,9.2)  
b = (2.3,-5.1)  
c = (7.8,4.1)  
c.equals(d): true  
c == d: false

CMCS 433, Spring 2003

24

## Downcasting

- **(Bar) foo**
  - run-time exception if object reference by **foo** is not a subclass of **Bar**
  - compile-time error if **Bar** is not a subtype of **foo** (i.e. it always throws an exception)
  - doesn't transform anything, just allows treating the result as if it were of type **Bar**
- **o instanceof Foo** returns true iff **o** is an instance of a subclass of **Foo**

CMCS 433, Spring 2003

25

## Different from C++

- No malloc()
  - Only new()
- No free()
  - Uses garbage collection
- No pointer operations: \*, &, ->, +, ++, etc.
  - Simplifies usage and implementation
- Method parameters pass-by-value
  - but object parameters are references to heap objects that can be changed

CMCS 433, Spring 2003

26

CMSC433, Fall 2002  
Programming Language  
Technology and Paradigms  
Basic Java

## Last Time

- OO principles
  - Keys: abstraction, encapsulation, sharing
- Java basics
  - Everything is an Object
    - Object “contract”
    - Downcasting
  - Objects are always referenced on the Heap

CMCS 433, Spring 2003

28

## Visibility Modifiers

- Indicate visibility of
  - Classes
  - Methods
  - Fields
- Support abstraction
  - Clients unaffected by change in implementation
- Support encapsulation
  - Prevents leaking of information to clients

CMCS 433, Spring 2003

29

## Class modifiers

- **public** – class visible outside package
- **final** – no other class can extend this class
- **abstract** – no instances of this class can be created
  - only instances of extensions of the class
- No modifier implies *package*-level scope

CMCS 433, Spring 2003

30

## Variable / method visibility

- **public** – visible everywhere
- **private** – visible only within this class
- **protected** – visible within same package or in subclass
- **package** (default) – visible within same package

CMCS 433, Spring 2003

31

## Instance vs. static variables

- **static** – the data is stored “with the class”
  - static variables allocated once, no matter how many objects created
  - static methods are not specific to any class instance, so can’t refer to **this** or **super**
- Can reference class variables and methods through either class name or an object ref
  - Clearer to reference via the class name

CMCS 433, Spring 2003

32

## Instance vs. static

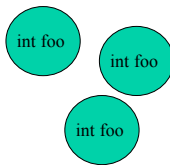
Class definition

```
Public class Foo {  
  int foo;  
  static int bar;  
}
```

Class implementation

```
Foo  
int bar;
```

Objects of class Foo



CMCS 433, Spring 2003

33

## Examples

- public **static** void main(String args[]) { ... }
- public class Math {  
 public final **static** PI = 3.14159...;  
}
- public class System {  
 public **static** PrintStream out = ...;  
}

CMCS 433, Spring 2003

34

## Instance variable modifiers

- **final** – can't be changed; must be initialized in declaration or in constructor
- **transient, volatile**  
– will cover later

CMCS 433, Spring 2003

35

## Method modifiers

- **final** – this method cannot be overridden
  - useful for security
  - allows compiler to inline method
    - good VM's will do this without the hint
- **abstract** – no implementation provided
  - class must be abstract
- **native, synchronized**  
– will cover later

CMCS 433, Spring 2003

36

## CMSC433, Spring 2003

### Basic Java (Part II)

William Pugh  
February 4th, 2003

## Administrivia

- See me if you are on the wait list
- Resolve any problems getting onto the cluster today
- Discussion on Thursday of procedure for submitting projects
- Discussion of quiz on Thursday
- Reading:
  - [Thinking in Java, Chapters 4 and 7](#)

CMCS 433, Spring 2003

38

## Method invocation

- Method names can be *overloaded*
  - method invoked is determined by both its name and the types of the parameters
  - resolved at compile-time, based on compile-time types
- Methods can also be *overridden*
  - define a method also defined by a superclass
  - arguments and result types must be identical
  - resolved at run-time, based on type of object method is invoked on

CMCS 433, Spring 2003

39

## Overriding

- Overriding
  - methods with same name and argument types in child class override method in parent class
  - you can override/hide instance variables
    - both variables will exist, but don't do it

```
class Parent {  
    int cost;  
    void add(int x) {  
        cost += x;  
    }  
}  
class Child extends Parent {  
    void add(int x) {  
        if (x > 0) cost += x;  
    }  
}
```

CMCS 433, Spring 2003

40

## Overloading

- Methods with the same name, but different parameters (count or types) are overloaded

```
class Parent {
    int cost;
    void add (int x) {
        cost += x;
    }
    void add (String s) throws NumberFormatException {
        cost += Integer.parseInt(s);
    }
}
```

CMCS 433, Spring 2003

41

## Dynamic Method Dispatch

- If you have a ref **a** of type **A** to an object that is actually of type **B** (a subclass of **A**)
  - instance methods invoked on **a** will get the methods for class **B** (like C++ virtual functions)
  - class methods invoked on **a** will get the methods for class **A**
    - invoking class methods on objects strongly discouraged

CMCS 433, Spring 2003

42

## Simple Dynamic Dispatch Example

```
public class A {
    String f() {return "A.f() "; }
    static String g() {return "A.g() "; }
}
public class B extends A {
    String f() {return "B.f() "; }
    static String g() {return "B.g() "; }
    public static void main(String args[]) {
        A a = new B();
        B b = new B();
        System.out.println(a.f() + a.g() +
                           b.f() + b.g());
    }
}
java B generates:
B.f() A.g() B.f() B.g()
```

CMCS 433, Spring 2003

43

## Self reference

- **this** refers to the object the method is invoked on
- **super** refers to the same object as **this**
  - but used to access methods/variables in superclass
- Like C++

CMCS 433, Spring 2003

44

## Constructors

- Declaration syntax same as C++
  - no return type specified
  - method name same as class
- First statement can be **this(args)** or **super(args)**
  - if those are omitted, **super()** is called
  - must be very first statement, even before variable declarations
- *not* used for type conversions or assignments
- void constructor generated if no constructors given

CMCS 433, Spring 2003

45

## Garbage collection

- Objects that are no longer accessible can be garbage collected
- Method **void finalize()** called when an object is collected
  - best to avoid using it, since no way to tell when it will get called
- Garbage collection not a major performance bottleneck
  - **new/delete** in C++ can be expensive too

CMCS 433, Spring 2003

46

## Detailed Example

- Shows
  - polymorphism for both method receiver and arguments
  - static vs. instance methods
  - overriding instance variables

CMCS 433, Spring 2003

47

## Source code for classes

```
class A {
    String f(A x) { return "A.f(A) "; }
    String f(B x) { return "A.f(B) "; }
    static String g(A x) { return "A.g(A) "; }
    static String g(B x) { return "A.g(B) "; }
    String h = "A.h";
    String getH() { return "A.getH(): " + h; }
}
class B extends A {
    String f(A x) { return "B.f(A)/ " + super.f(x); }
    String f(B x) { return "B.f(B)/ " + super.f(x); }
    static String g(A x) { return "B.g(A) "; }
    static String g(B x) { return "B.g(B) "; }
    String h = "B.h";
    String getH() {
        return "B.getH(): " + h + "/" + super.h;
    }
}
```

CMCS 433, Spring 2003

48

```

A a = new A(); A ab = new B(); B b = new B();
System.out.println( a.f(a) + a.f(ab) + a.f(b) );
// A.f(A) A.f(A) A.f(B)
System.out.println( ab.f(a) + ab.f(ab) + ab.f(b) );
// B.f(A)/A.f(A) B.f(A)/A.f(A) B.f(B)/A.f(B)
System.out.println( b.f(a) + b.f(ab) + b.f(b) );
// B.f(A)/A.f(A) B.f(A)/A.f(A) B.f(B) A.f(B)

System.out.println( a.g(a) + a.g(ab) + a.g(b) );
// A.g(A) A.g(A) A.g(B)
System.out.println( ab.g(a) + ab.g(ab) + ab.g(b) );
// A.g(A) A.g(A) A.g(B)
System.out.println( b.g(a) + b.g(ab) + b.g(b) );
// B.g(A) B.g(A) B.g(B)

System.out.println( a.h + " " + a.getH() );
// A.h A.getH():A.h
System.out.println( ab.h + " " + ab.getH() );
// A.h B.getH():B.h/A.h
System.out.println( b.h + " " + b.getH() );
// B.h B.getH():B.h/A.h

```

## Invocation and results

49

## What to notice

- Invoking **ab.f(ab)** invokes **B.f(A)**
  - run-time type of object determines method invoked
  - compile-time type of arguments used
- **ab.h** gives the **A** version of **h**
- **ab.getH()**
  - **B.getH()** method invoked
  - in **B.getH()**, **h** gives **B** version of **h**
- Use of **super** in class **B** to reach **A** version of methods/variables
- **super** not allowed in static methods

CMCS 433, Spring 2003

50

## Interfaces

- An interface lists supported methods
  - No constructors or implementations allowed
  - Can have final static variables
- A class can *implement* (be a subtype of) one or more interfaces
- Using the name of an interface as a type (i.e. to declare a variable) means
  - a reference to any instance of a class that implements the interface is a permitted value
  - **null** is also allowed

CMCS 433, Spring 2003

51

## Interface example

```

public interface Comparable {
    public int compareTo(Object o)
}
public class Util {
    public static void sort(Comparable [] ) { ... }
}
public class Choices implements Comparable {
    public int compareTo(Object o) {
        return ... ;
    }
}
...
    Choices [] options = ... ;
    Util.sort(options);
...

```

CMCS 433, Spring 2003

52

## No multiple inheritance

- A class type can be a subtype of many other types (**implements**)
- But can only inherit method implementations from one superclass (**extends**)
- Not a big deal
  - multiple inheritance rarely, if ever, necessary and often badly used
- And it's complicated to implement well

CMCS 433, Spring 2003

53

## Poor man's polymorphism

- Every object is an **Object**
- Thus, a data structure **Set** that implements sets of **Objects**
  - can summarily hold **Strings**
  - or images
  - or ... anything!
- The trick is getting them back out:
  - When given an **Object**, you have to downcast it

CMCS 433, Spring 2003

54

## Example

```
class DumbSet {
    public void insert(Object o) {...}
    public bool member(Object o) {...}
    public Object any() {...}
}

class MyProgram {
    public static void main(String[] args) {
        DumbSet set = new DumbSet();
        String s1 = "foo";
        String s2 = "bar";
        set.insert(s1);
        set.insert(s2);
        System.out.println(s1+"in set?" + set.member(s1));
        String s = (String)set.any(); // downcast
        System.out.println("got "+s);
    }
}
```

CMCS 433, Spring 2003

55

## Wrapper classes

- To create **Integer**, **Boolean**, **Double**, ...
  - that is a subclass of **Object**
  - useful/required for polymorphic methods
    - **HashMap**, **LinkedList**, ...
  - used in reflection classes
- Include many utility functions
  - e.g., convert to/from **String**
- **Number**: superclass of **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**
  - allows conversion to any other numeric primitive type

CMCS 433, Spring 2003

56

## Class Objects

- For each class, there is an object of type **Class**
- Describes the class as a whole
  - used extensively in Reflection package
- **Class.forName(“MyClass”)**
  - returns class object for **MyClass**
  - will load **MyClass** if needed
- **Class.forName(“MyClass”).newInstance()**
  - creates a new instance of **MyClass**
- **MyClass.class** gives the **Class** object for **MyClass**

CMCS 433, Spring 2003

57

## Array types

- If **S** is a subtype of **T**
  - **S[]** is a subtype of **T[]**
- **Object[]** is a supertype of all arrays of reference types
- Arrays elements must match array allocation type
  - Storing into an array generates a run-time check that the type stored is a subtype of the allocation type of the array elements

CMCS 433, Spring 2003

58

## Example: Object[]

```
public class TestArrayTypes {
    public static void reverseArray(Object [] A) {
        for(int i=0, j=A.length-1; i<j; i++,j--) {
            Object tmp = A[i];
            A[i] = A[j];
            A[j] = tmp;
        }
    }
    public static void main(String [] args) {
        reverseArray(args);
        for(int i=0; i < A.length; i++)
            System.out.println(args[i]);
    }
}
```

CMCS 433, Spring 2003

59

CMSC433, Spring 2003

Exceptions

## Exceptions

- On an error condition, we *throw* an exception
- At some point up the call chain, the exception is *caught* and the error is handled
- Separates normal from error-handling code
- A form of non-local control-flow
  - Like `goto`, but structured

CMCS 433, Spring 2003

61

## Throwing an Exception

- Create a new object of the class `Exception`, and **throw** it

```
if (i > 0 && i < a.length) {
    return (a[i])
}
else throw new ArrayIndexOutOfBoundsException();
```
- Exceptions thrown are part of return type
  - when overriding a method in a superclass
  - can't throw anything that would surprise a superclass object

CMCS 433, Spring 2003

62

## Method throws declarations

- A method declares the exceptions it might throw
  - `public void openNext() throws`  
`UnknownHostException,`  
`EmptyStackException`  
`{ ... }`
- Must declare any exception the method *might* throw
  - unless it is caught in the method
  - includes exceptions thrown by called methods
  - certain built-in exceptions excluded

CMCS 433, Spring 2003

63

## Exception Handling

- All exceptions eventually get caught
- First **catch** with supertype of the exception catches it
- **finally** is always executed

```
try { if (i == 0) return; myMethod(a[i]); }
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("a[] out of bounds"); }
catch (MyOwnException e) {
    System.out.println("Caught my error"); }
catch (Exception e) {
    System.out.println("Caught" + e.toString()); throw e; }
finally { /* stuff to do regardless of whether an exception */
        /* was thrown or a return taken */ }
```

CMCS 433, Spring 2003

64

## java.lang.Throwable

- Exception is a subclass of **Throwable**
- Many objects of class **Throwable** have a message
  - specified when constructed, as **String**
  - **String getMessage()** returns the message
- **String toString()**
- **void printStackTrace()**
- **void printStackTrace(PrintWriter s)**

CMCS 433, Spring 2003

65

## Example Application

```
public class BufferedReader {
    public String readLine() throws IOException { ... } ...
}
public class Echo {
    public static void main(String args[]) {
        BufferedReader in = ...
        try {
            while((s = in.readLine()) != null)
                System.out.println(s);
        } catch(IOException e) {
            System.out.println(e.stackTrace());
        }
    }
}
```

CMCS 433, Spring 2003

66

## The easy way

```
public class BufferedReader {
    public String readLine() throws IOException { ... } ...
}
public class Echo {
    public static void main(String args[]) throws Exception {
        BufferedReader in = ...
        while((s = in.readLine()) != null)
            System.out.println(s);
    }
}
```

CMCS 433, Spring 2003

67

## Creating New Exceptions

- User-defined exception is just a class that is a subclass of **Exception**

```
class MyOwnException extends Exception {}
class MyClass {
    void oops() throws MyOwnException {
        if (some_error_occurred) {
            throw new MyOwnException();
        }
    }
}
```

CMCS 433, Spring 2003

68

## Java Libraries

## You should familiarize yourself

- Packages
  - java.lang
  - java.util
  - java.net
  - java.io
- Read the documentation on line

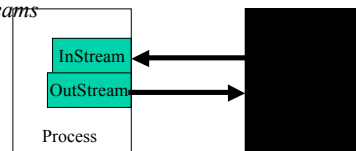
CMCS 433, Spring 2003

70

## I/O and Network Libraries

## I/O streams

- Raw communication takes place using byte *streams*



- Java also provides *readers* and *writers*
  - *character streams*
- Applies to files, network connections, strings, etc.

CMCS 433, Spring 2003

72

## I/O Classes

- **OutputStream** – byte stream going out
- **Writer** – character stream going out
- **InputStream** – byte stream coming in
- **Reader** – character stream coming in

CMCS 433, Spring 2003

73

## OutputStream - bytes

- Example classes
  - `ByteArrayOutputStream` – goes to byte []
  - `FileOutputStream` – goes to file
- Wrappers – wrapped around `OutputStream`
  - `BufferedOutputStream`
  - `ObjectOutputStream` – serialization of object graph

CMCS 433, Spring 2003

74

## Writer - characters

- `OutputStreamWriter`
  - wraps around `OutputStream` to get a `Writer`
  - takes characters, converts to bytes
  - can specify encoding used to convert
- Other wrappers
  - `PrintWriter` – supports `print`, `println`
  - `BufferedWriter`
- Other Writers
  - `CharArrayWriter`
  - `StringWriter`

CMCS 433, Spring 2003

75

## InputStream - bytes

- Example classes
  - `ByteArrayInputStream`
  - `FileInputStream`
- Wrappers – wrapped around `InputStream`
  - `BufferedInputStream`
  - `PushedBackInputStream`

CMCS 433, Spring 2003

76

## Reader - characters

- **InputStreamReader**
  - wrap around **InputStream** to get a **Reader**
  - takes bytes, converts to characters
  - can specify encoding used to convert
- Other wrappers
  - **BufferedReader** – efficient, supports **readLine()**
    - **LineNumberReader** – reports line numbers
  - **PushBackReader**
- Other Readers
  - **CharArrayReader**
  - **StringReader**

CMCS 433, Spring 2003

77

## Applications and I/O

- Java “external interface” is a public class
- via **public static void main(String [] args)**
- **args[0]** is first argument
  - unlike C/C++
- **System.out** and **System.err** are **PrintStreams**
  - should be **PrintWriter**, but would break 1.0 code
  - **System.out.print(...)** prints a string
  - **System.out.println(...)** prints a string with a newline
- **System.in** is an **InputStream**
  - not quite so easy to use

CMCS 433, Spring 2003

78

## Input (JDK 1.1 and higher)

- Wrap **System.in** in an **InputStreamReader**
  - converts from bytes to characters
- Wrap the result in a **BufferedReader**
  - makes input operations efficient
  - supports **readLine()** interface
- **readLine()** returns a string
  - returns **null** if at EOF

CMCS 433, Spring 2003

79

## Example Echo Application

```
import java.io.*;
public class Echo {
    public static void main(String [] args) throws Exception {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        int i = 1;
        String s;
        while((s = in.readLine()) != null)
            System.out.println((i++) + ": " + s);
    }
}
```

CMCS 433, Spring 2003

80

## Java Networking

- class `Socket`
  - Client-side connections to servers
- class `ServerSocket`
  - Server-side “listen” socket
  - Awaits and responds to connection requests

CMCS 433, Spring 2003

81

## Example Client/Server

```
ServerSocket s = new ServerSocket(5001);  
Socket conn = s.accept();  
InputStream in = conn.getInputStream();  
OutputStream out = conn.getOutputStream();
```

*Server code*

server

client

*Client code*

```
Socket conn = new Socket("www.cs.umd.edu", 5001);  
InputStream in = conn.getInputStream();  
OutputStream out = conn.getOutputStream();
```

CMCS 433, Spring 2003

82

## Example Client/Server

```
ServerSocket s = new ServerSocket(5001);  
Socket conn = s.accept();  
InputStream in = conn.getInputStream();  
OutputStream out = conn.getOutputStream();
```

*Server code*

server



client

```
Socket conn = new Socket("www.cs.umd.edu", 5001);  
InputStream in = conn.getInputStream();  
OutputStream out = conn.getOutputStream();
```

*Client code*

CMCS 433, Spring 2003

83

## Example Client/Server

```
ServerSocket s = new ServerSocket(5001);  
Socket conn = s.accept();  
InputStream in = conn.getInputStream();  
OutputStream out = conn.getOutputStream();
```

*Server code*

server



client

?

```
Socket conn = new Socket("www.cs.umd.edu", 5001);  
InputStream in = conn.getInputStream();  
OutputStream out = conn.getOutputStream();
```

*Client code*

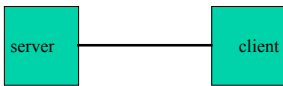
CMCS 433, Spring 2003

84

## Example Client/Server

```
ServerSocket s = new ServerSocket(5001);
Socket conn = s.accept();
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

*Server code*



*Note: The server can still accept other connection requests on port 5001*

```
Socket conn = new Socket("www.cs.umd.edu", 5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

*Client code*

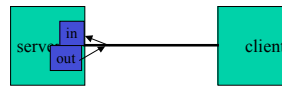
CMCS 433, Spring 2003

85

## Example Client/Server

```
ServerSocket s = new ServerSocket(5001);
Socket conn = s.accept();
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

*Server code*



```
Socket conn = new Socket("www.cs.umd.edu", 5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

*Client code*

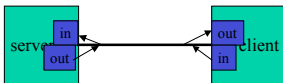
CMCS 433, Spring 2003

86

## Example Client/Server

```
ServerSocket s = new ServerSocket(5001);
Socket conn = s.accept();
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

*Server code*



```
Socket conn = new Socket("www.cs.umd.edu", 5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

*Client code*

CMCS 433, Spring 2003

87

## Possible Failures

- Server-side
  - ServerSocket port already in use
  - Client dies after accept
- Client-side
  - Server dies
  - No one listening on port
- In all cases IOException thrown
  - Must use appropriate throw/try/catch constructs

CMCS 433, Spring 2003

88

## Utility Libraries

## java.util

- Lots of stuff
  - Lists
  - Maps
  - Sets
  - Iterators

CMCS 433, Spring 2003

90

## Other libraries

- java.lang.Math
  - abstract final class – only static members
  - includes constants  $e$  and  $\pi$
  - includes static methods for trig, exponentiation, min, max, ...
- java.text
  - text formatting tools
    - class **MessageFormat** provides printf/scanf functionality
  - lots of facilities for internationalization

CMCS 433, Spring 2003

91

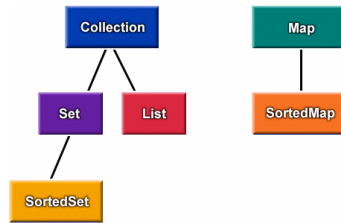
## Java Container Classes

- A unified architecture for representing and manipulating collections of objects
- Container classes contain three things:
  - Interfaces: abstract data types representing collections of objects
  - Implementations: concrete implementations of the collection interfaces
  - Algorithms: methods that perform computations on objects that implement collection interfaces

CMCS 433, Spring 2003

92

## Container class hierarchy



CMCS 433, Spring 2003

93

## Collection Classes

- Collections contain groups of objects (elements)
- Collection interface is not implemented in Java. Subinterfaces implemented
  - Set: unordered, can't contain duplicate elements
    - HashSet – unordered, no duplicates
    - TreeSet – ordered by value, no duplicates
  - List: ordered by position, can contain duplicate elements
    - LinkedList – unordered, dynamic size, add/delete quick
    - ArrayList – unordered, dynamic size, random access quick

CMCS 433, Spring 2003

94

## Map Classes

- A Map is an object that contains key:value pairs
- Maps cannot contain duplicate keys:
  - Each key can map to at most one value
- Map not implemented. Subinterfaces implemented
  - HashMap, entries stored in a hash table
  - TreeMap, entries maintained in sorted order
- Variants
  - Ordered/unordered (e.g., map vs. sorted map)

CMCS 433, Spring 2003

95

## Object Ordering

- Two ways to order objects:
  - Comparable interface provides automatic *natural order* on classes that implement it
  - Comparator interface gives the programmer complete control over object ordering

CMCS 433, Spring 2003

96

## compareTo Interface

- `public int compareTo(Object o)`
- The natural comparison method (i.e., default)
  - Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than o
  - `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))`
    - implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.
  - `(x.compareTo(y)>0 && y.compareTo(z)>0) => x.compareTo(z)>0.`
  - `x.compareTo(y)==0 => sgn(x.compareTo(z)) == sgn(y.compareTo(z))`
  - Recommended that `(x.compareTo(y)==0) == (x.equals(y))`

CMCS 433, Spring 2003

97

## Comparator Interface

- When natural order isn't acceptable
- `public int compare(Object o1, Object o2)`
  - Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second
  - `sgn(compare(x, y)) == -sgn(compare(y, x))`
    - (This implies that `compare(x, y)` must throw an exception if and only if `compare(y, x)` throws an exception.)
  - `((compare(x, y)>0) && (compare(y, z)>0)) => compare(x, z)>0.`
  - `compare(x, y)==0 => sgn(compare(x, z))==sgn(compare(y, z))`
  - recommended `(compare(x, y)==0) == (x.equals(y))`
- `public boolean equals(Object obj)`
  - Indicates whether some other object is "equal to" this Comparator.

CMCS 433, Spring 2003

98

## Example1

```
import java.util.*;
import java.awt.*;
class MyPoint extends java.awt.Point
implements Comparable {
    MyPoint(int x, int y) (super(x,y));
    public int compareTo(Object o) {
        MyPoint p = (MyPoint)o;
        double d1 = Math.sqrt(x*x + y*y);
        double d2 = Math.sqrt(p.x*p.x +
            p.y*p.y);
        if (d1 < d2) {return -1;}
        else if (d2 < d1) {return 1;}
        return 0;
    }
}

class Sort3 {
    public static void main(String[] args)
    {
        Random rnd = new Random();
        MyPoint[] points = new MyPoint[10];
        for (int i=0; i<points.length; i++) {
            points[i] = new MyPoint
                (rnd.nextInt(100),rnd.nextInt(100));
            System.out.println(points[i]);
        }
        System.out.println("-----");
        Arrays.sort(points);
        //Print the points
        for (int i=0; i<points.length; i++){
            System.out.println(points[i]);
        }
    }
}
```

CMCS 433, Spring 2003

99

## Example2

```
import java.util.*;
import java.awt.*;
class MyPoint extends
java.awt.Point implements
Comparable {
    MyPoint(int x, int y) {
        super(x, y);
    }
    public int
compareTo(Object o) {
        MyPoint p = (MyPoint)o;
        return x - p.x;
    }
}

class Sort2 {
    public static void main(String[]
args) {
        Random rnd = new Random();
        MyPoint[] points = new MyPoint[10];
        for (int i=0; i<points.length; i++) {
            points[i] = new MyPoint
                (rnd.nextInt(100), rnd.nextInt(100));
            System.out.println(points[i]);
        }
        System.out.println("-----");
        Arrays.sort(points);
        //Print the points
        for (int i=0; i<points.length; i++){
            System.out.println(points[i]);
        }
    }
}
```

CMCS 433, Spring 2003

100

## Output

Sort2 ..... // after sort

MyPoint[x=1,y=95]

MyPoint[x=2,y=16]

MyPoint[x=3,y=26]

MyPoint[x=12,y=95]

MyPoint[x=22,y=55]

MyPoint[x=30,y=73]

MyPoint[x=31,y=42]

MyPoint[x=66,y=33]

MyPoint[x=70,y=33]

MyPoint[x=80,y=31]

Sort3 ..... // after sort

MyPoint[x=2,y=0]

MyPoint[x=0,y=15]

MyPoint[x=18,y=4]

MyPoint[x=39,y=13]

MyPoint[x=39,y=19]

MyPoint[x=42,y=23]

MyPoint[x=65,y=5]

MyPoint[x=38,y=74]

MyPoint[x=80,y=40]

MyPoint[x=87,y=62]