

## CMSC433, Spring 2004 Programming Language Technology and Paradigms

### Java Review

Jeff Foster  
February 5, 2004

## Administrivia

- Project submission instructions posted on-line
  - Project due Wednesday
- Reading: Liskov ch 3, 9, 10

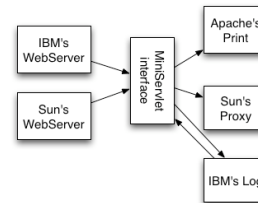
67

## Project 1 Notes

- Project 1 shows some of the issues in software components
  - And a few design patterns as well (more later)
- MiniServlet is the interface between components
  - WebServer shouldn't assume anything about servlets it will run
  - MiniServlets shouldn't assume anything about server that will invoke them

68

## Multiple Vendors



69

## Things to Check

- If “grep Proxy WebServer.java” finds a match, you likely have a problem
- Can you put just WebServer.java and MiniServlet.java in a directory and compile them?
- Can you put everything except WebServer.java in a directory and compile them all?
- Can you use TestServlet to run your servlets?

70

## 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

71

## 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 printable representation
public int hashCode() – key for hash table
public void finalize() – called if object is garbage collected
```

- And others ...

72

## Equality

- Object .equals(Object) method
  - Structural (“conceptual”) equality
- == operator (!= as well)
  - True if arguments reference the same object
  - **o == p** implies **o.equals(p)**

73

## Overriding Equals

```
class Foo {
    public boolean equals(Foo f) { ... } // wrong!
}

class Foo {
    public boolean equals(Object o) { // right!
        if (!(o instanceof Foo))
            return false;
        ...
    }
}
```

The first case creates an *overloaded* method, while the second *overrides* the parent (**Object**) method.

74

## Overriding hashCode

- **hashCode()** is used for objects that may be stored in hash table
- Rule of thumb: If you override **equals()** or **hashCode()**, you should also override the other
  - **a.equals(b)** implies **a.hashCode() == b.hashCode()**

75

## Preconditions

- Functions often have requirements on their inputs

```
// Return maximum element in A[i..j]
int findMax(int[] A, int i, int j) { ... }
```

- A is non-empty
- i and j must be non-negative
- i and j must be less than A.length
- i < j (maybe)

- These are called *preconditions* or *requires* clauses

76

## Dealing with Errors

- What do you do if a precondition isn't met?
- What do you do if something unexpected happens?
  - Try to open a file that doesn't exist
  - Try to write to a full disk

77

## Signaling Errors

- Style 1: Return invalid value

```
// Returns value key maps to, or null if no
// such key in map
Object get(Object key);
```

- Disadvantages?

78

## Signaling Errors (cont'd)

- Style 2: Return an invalid value and status

```
static int lock_rdev(mdk_rdev_t *rdev) {
    ...
    if (bdev == NULL)
        return -ENOMEM;
    ...
}

// Returns NULL if error and sets global
// variable errno
FILE *fopen(const char *path, const char *mode);
```

79

## Problems with These Approaches

- What if all possible return values are valid?
  - E.g., `findMax` from earlier slide
- What if client forgets to check for error?
  - No compiler support
- What if client can't handle error?
  - Needs to be dealt with at a higher level

80

## Exceptions in Java

- 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

81

## 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 method in superclass, cannot throw any more exceptions than parent's version

82

## 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 (masked by) the method
  - Includes exceptions thrown by called methods
  - Certain kinds of exceptions excluded

83

## 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 */ }
```

84

## Masking Exceptions

- Handle exception and continue

```
while ((s = ...) != null) {
    try {
        FileInputStream f =
            new FileInputStream(s);
        ...
    }
    catch (FileNotFoundException e) {
        System.out.println(s + " not found");
    }
}
```

85

## Reflecting Exceptions

- Pass exception up to higher level
  - Automatic support for throwing same exception
  - Sometimes useful to throw different exception

```
try {
    ... a[5] ...
}
catch (IndexOutOfBoundsException e) {
    throw new EmptyException("Arrays.min");
}
```

86

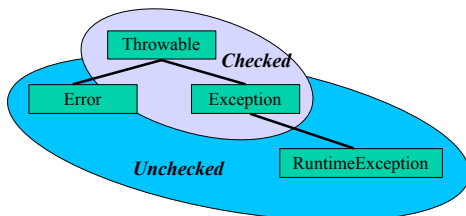
## Exception Chaining

- Indicate the cause of a thrown exception
  - Specify the exception that caused this one
  - Shows cause chain in stack trace

```
try {
    ... a[0] ...
}
catch (IndexOutOfBoundsException e) {
    // e can be retrieved from exn with getCause()
    throw new Exception("Arrays.min", e);
}
```

87

## Exception Hierarchy



88

## Unchecked Exceptions

- Subclasses of **RuntimeException** and **Error** are unchecked
  - Need not be listed in method specifications
- Currently used for things like
  - NullPointerException
  - IndexOutOfBoundsException
  - VirtualMachineError
- Is this a good design?

89