CMSC433, Spring 2005
Programming Language Technology and Paradigms

Java Review

Jeff Foster
January 27, 2005

Administrivia

- Project 1 will be posted later today
  - Due February 9
- Supplemental reading: Eckel ch. 1, 7, 8, 9

Java

- Descended from Simula67, SmallTalk, others
  - Superficially similar to C, C++
- Fully specified, compiles to virtual machine
  - Machine-independent
- Secure
  - Bytecode verification ("type-safe")
  - Security manager

Object Orientation

- Combining data and behavior
  - Objects, not developers, decide how to carry out operations
- Sharing via abstraction and inheritance
  - 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

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);
  }
}
```


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;
    c = a.plus(b);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
}
```


The Class Hierarchy

- Classes by themselves are a powerful tool
  - Support *abstraction* and *encapsulation*
- Java also provides two other abilities
  - *Interfaces* allow different classes to be treated the same
  - *Inheritance* allows code reuse
    - Note: When you inherit from a class, you also "implement" the class's "interface"

Project 1: Interfaces

```
public interface MiniServlet extends Runnable {
    void setArg(String arg);
    void setOutputStream(OutputStream out);
}

class HelloWorld implements MiniServlet { ... }
class Print implements MiniServlet { ... }

MiniServlet s = new HelloWorld();
if (...) s = new Print();
s.setArg(...);
```

Interfaces

- An interface lists supported (public) methods
  - No constructors or implementations allowed
  - Can have final static variables
- A class can implement (be a subtype of) zero or more interfaces
- Given some interface **I**, declaring **I x =** ... means
  - **x** must refer to an instance of a class that implements **I**, or else **null**

Interface Inheritance

- Interfaces can *extend* other interfaces
  - Sometimes convenient form of reuse (see project 1)
- Given interfaces **I1** and **I2** where **I2** extends **I1**
  - If **C** implements **I2**, then **C** implements **I1**
- Since a class can implement multiple interfaces, interface extensions are often not needed

Inheritance

- Each Java class *extends* or inherits code from exactly one superclass
- Permits reusing classes to define new objects
  - Can define the behavior of the new object in terms of the old one

Example

```
class Point {
  int getX() { ... }
  int getY() { ... }
}
class ColorPoint extends Point {
  int getColor() { ... }
}
```

- **ColorPoint** reuses **getX()** and **getY()** from **Point**
- **ColorPoint** “implements” the **Point** “interface”
  - They can be used anywhere a **Point** can be

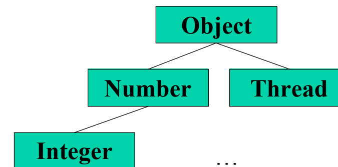
Subtyping

- Both inheritance and interfaces allow one class to be used where another is specified
  - This is really the same idea: subtyping
- We say that **A** is a *subtype* of **B** if
  - **A** extends **B** or a subtype of **B**, or
  - **A** implements **B** or a subtype of **B**

13

Java Design

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



…

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

14

No Multiple Inheritance

- A class type can implement many interfaces
- But can only extend one superclass
- Not a big deal
  - Multiple inheritance rarely, if ever, necessary and often badly used
  - And it's complicated to implement well

15

Abstract Classes

- Sometimes want a class with some code, but with some methods unwritten
  - It can't be an interface because it has code
  - It can't be a regular class because it doesn't have all the code
    - You can't instantiate such a class
- Instead, we can mark such a class as *abstract*
  - And mark the unimplemented methods as *abstract*

16

Example from JDK

```
public abstract class OutputStream {
  public abstract void write(int b) ...;
  public void write(byte b[], int off, int len) ... {
      ... write(b[off + i]);...
  }
  ...
}
```

- Subclasses of OutputStream need not override the second version of write(...)
  - But they do need to override the first one, since it's abstract
  - (Note: They may want to override anyhow for efficiency)

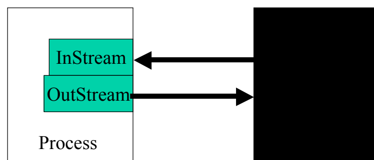
Example from Project 1

```
public abstract class ServletFilter extends
   OutputStream implements MiniServlet
{
    public abstract void write(int b) ...;
    public abstract void close() ...;
    public abstract void flush() ...;

    public void setArg(String arg);
    public void setOutputStream(OutputStream out);
}
```


I/O streams

- Raw communication takes place using streams



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

I/O Classes

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

Some OutputStreams and Writers

- Example classes
  - **ByteArrayOutputStream** – goes to byte []
  - **FileOutputStream** – goes to file
- **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**
  - **StringWriter**

Applications and I/O

- Java "external interface" is a public class
  - **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

Java Networking

- class Socket
  - Communication channel
- class ServerSocket
  - Server-side "listen" socket
  - Awaits and responds to connection requests

Example Client/Server

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

Server code

server

client

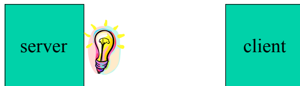
```
Socket conn = new Socket ("www.cs.umd.edu", 5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```

Client code

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();
```

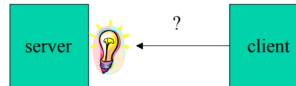
Client code

25

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();
```

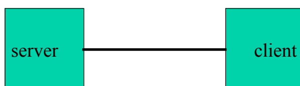
Client code

26

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();
```

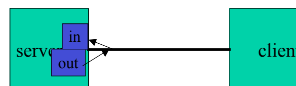
Client code

27

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();
```

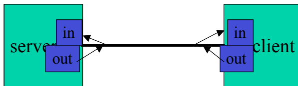
Client code

28

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

29

Possible Failures

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

30

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**

31

CMSC433, Spring 2005 Programming Language Technology and Paradigms

Java Review

Jeff Foster

Feburary 1, 2005

Administrivia

- Supplemental reading: Eckel, ch 8, 9
- Class accounts e-mailed out
  - Contact me if you didn't get one

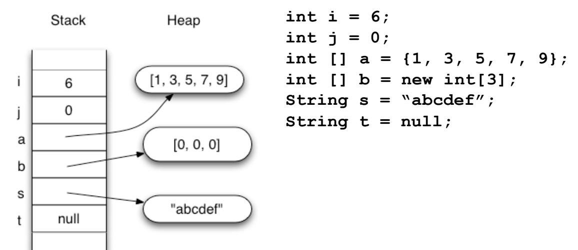
Demo

- Running basic web server
  - Finding hostname of machine
  - Connecting with web browser
- Telnet to a web server

Objects and Variables

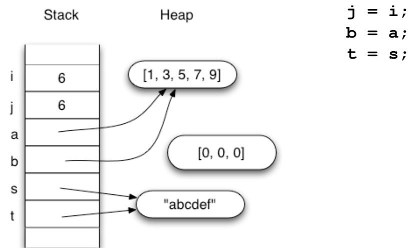
- Variables of a primitive type contain *values*
  - E.g., byte, char, int, ...
  - int i = 6;
  - Uninitialized fields that are values contain 0
  - Assignment copies values
- Variables of other types contain *references* to the heap
  - int[] a = new int[3];
  - Objects are allocated with new
  - Uninitialized fields that are object references are null
  - *Assignment copies references*, not the objects themselves

Example




```
int i = 6;
int j = 0;
int [] a = {1, 3, 5, 7, 9};
int [] b = new int[3];
String s = "abcdef";
String t = null;
```

Example: Assignments




```
j = i;
b = a;
t = s;
```

37

Garbage Collection

- What happens to array [0, 0, 0] in previous example?
  - It is no longer accessible
  - When Java performs *garbage collection* (GC) it will automatically reclaim the memory it uses
- Notice: No free() or delete in Java
  - Makes it much easier to write correct programs
  - Most of the time, very efficient

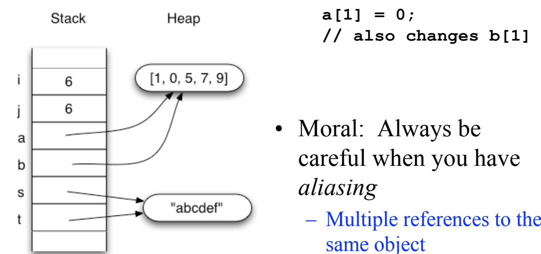
38

Mutability

- An object is mutable if its state can change
  - Example: Arrays are mutable
- An object is immutable if its state never changes
  - Once its been initialized
  - Example: Strings in Java are not mutable
    - There are no methods to change the state of a String

39

Example: Mutability




```
a[1] = 0;
// also changes b[1]
```

- Moral: Always be careful when you have *aliasing*
  - Multiple references to the same object

40

Method Invocation

- Syntax

  o.m(arg1, arg2, ..., argn);

  – Run the **m** method of object **o** with arguments **arg1...argn**

- Two ways to reuse method names:
  – Methods can be overridden
  – Methods can be overloaded

41

Overriding

- Define a method also defined by a superclass

```
class Parent {
  int cost;
  void add(int x) {
    cost += x;
  }
}
```

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

42

Overriding (cont'd)

- Method with same name and argument types in child class overrides method in parent class
- Arguments and result types must be identical
  – Otherwise you are overloading the method
- Must raise the same or fewer exceptions
- Can override/hide instance variables
  – Both variables will exist, but don't do it

43

Declared vs. Actual Types

- The *actual type* of an object is its allocated type
  – **Integer o = new Integer(1);**
- A *declared type* is a type at which an object is being viewed
  – **Object o = new Integer(1);**
  – **void m(Object o) { ... }**
- Each object always has *one* actual type, but can have *many* declared types

44

Method Dispatch

- Consider again

  o.m(arg1, arg2, ..., argn);

- Only compiles if **o**'s declared type contains an appropriate **m** method

- Method corresponding to **o**'s actual type is what is invoked

45

Dynamic Dispatch Example

```
public class A {
   String f() { return "I'm an A! "; }
}

public class B extends A {
   String f() { return "I'm a B! "; }
   public static void main(String args[]) {
      A a = new B();
      B b = new B();
      System.out.println(a.f() + b.f());
   }
}
```

Prints I'm a B! I'm a B!

46

Self Reference

- **this** refers to the object the method is invoked on
  - Thus can access fields of this object as this.x or this.y
  - But more concise to omit

- **super** refers to the same object as **this**
  - But used to access methods/variables in superclass

47

Example of super

- Call a superclass method from a subclass

```
class Parent {
  int cost;
  void add(int x) {
    cost += x;
  }
}
```

```
class Child extends Parent {
  void add(int x) {
    if (x > 0) super.add(x);
  }
}
```

48

Overloading

- Methods with the same name, but different parameters (count or types) are overloaded
  - Invocation determined by name and types of params
  - Not return value or exceptions
- Resolved at **compile-time**, based on declared types
- Be careful: Easy to inadvertently overload instead of override!

Overloading Example

```
class Parent {
  int cost;
  void add(int x) {
    cost += x;
  }
  void add(Object s) throws NumberFormatException {
    cost += Integer.parseInt((String)s);
  }
}
class Child extends Parent {
  void add(String s) throws NumberFormatException {
    int x = Integer.parseInt(s);
    if (x > 0) cost += x;
  }
}
```

```
Child c = new Child();
c.add((Object)"-1");
System.out.println(c.cost);
```

Prints -1

Static Fields and Methods

- *static* – stored “with the class”
  - Static fields allocated once, no matter how many objects created
  - Static methods are not specific to any class instance, so cannot 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

Static Field Example

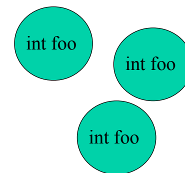
Class definition

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

Class implementation

Foo
int bar

Objects of class Foo



Some Static Fields and Methods

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

Static Method Dispatch

- Let **B** be a subclass of **A**, and suppose we have
  A a = new B(); *Declared type **A***
  *Actual type **B***
- Then
  - Class (static) methods invoked on a will get the methods for the *declared* type **A**
    - Invoking class methods via objects strongly discouraged
    - Instead, invoke through class
      - **A.m()** instead of **a.m()**

Static Method Dispatch Example

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

Prints This is A! This is B!

Better Use of Static Methods

```
public class A {
   static String g() { return "This is A! "; }
}
public class B extends A {
   static String g() { return "This is B! "; }
   public static void main(String args[]) {


      System.out.println(A.g() + B.g());
   }
}
```

Prints This is A! This is B!

Other Field Modifiers

- *final* – can't be changed
  - Must be initialized in declaration or in constructor
- *transient*, *volatile*
  - Will cover later
- *public*, *private*, *protected*, *package* (default)
  - Respectively, visible everywhere, only within this class, within same package or subclass, within same package

57

Other Method Modifiers

- *final* – this method cannot be overridden
  - Useful for security
  - Allows compiler to inline method
- *abstract* – no implementation provided
  - Class must be abstract
- *public* – visible outside this package
- *native*, *synchronized*
  - Will cover later

58

Poor Man's Polymorphism

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

59

Downcasting

- **(Bar) foo**
  - Run-time exception if object reference by **foo** is not a subtype of **Bar**
  - Compile-time error if **Bar** is not a subtype of **foo** (i.e., it always throws an exception)
  - No effect at run-time; just treats the result as if it were of type **Bar**
- **o instanceof Foo** returns true iff **o** is a subtype of **Foo**

60

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);
  }
}
```


Wrapper (Boxed) classes

- **Integer**, **Boolean**, **Double**, …
  - Are subclasses of **Object**
  - Useful/required for polymorphic methods
    - **HashTable**, **LinkedList**, …
  - Used in reflection classes
- Include many utility functions
  - For example, convert to/from **String**

Array Types

- Misfeature: suppose **S** is a subtype of **T**
  - then **S[]** is a subtype of **T[]**
- **Object[]** is a supertype of all arrays of reference types

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]);
  }
}
```

Problem with Subtyping Arrays

```
public class A { ... }
public class B extends A { void newMethod(); }
...
  void foo(void) {
     B[] bs = new B[];
     A[] as;

     as = bs;            // Since B[] subtype of A[]
     as[0] = new A();    // (1)
     bs[0].newMethod();  // (2)
  }
```

- Program compiles without warning
- Java must generate run-time check at (1) to prevent (2)
  - Type written to array must be subtype of declared type

CMSC433, Spring 2005 Programming Language Technology and Paradigms

Java Review

Jeff Foster

February 3, 2005

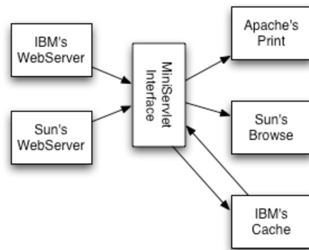
Administrivia

- Remember to check the newsgroup
  - Answers to a couple of common questions have been posted there
- Try submitting something for project 1
  - Make sure there are no problems with the server
  - E.g., submit the p1 files we gave you

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

Multiple Vendors



69

Things to Check

- If "grep Print 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;
    Foo other = (Foo) o;
    …
  }
}
```

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

Example

```
public class MyString {
  private String s;
  public MyString(String s) { this.s = s; }
  public boolean equals(Object o) {
    if (!(o instanceof String)) return false;
    return s.equals(((MyString) o).s);
  }
  public static void main(String[] args) {
    Set s = new HashSet();
    s.add(new MyString("Maryland"));
    System.out.println(
      s.contains(new MyString("Maryland")));
  }
}
```

(Based on Josh Bloch, Programming Puzzlers, Java One 2002)

76

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

77

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

78

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?

79

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);
```

80

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

81

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

82

Throwing an Exception

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

```
if (i >= 0 && i < a.length )
  return a[i];
throw new ArrayIndexOutOfBounds();
```

- Exceptions thrown are part of return type
  - When overriding method in superclass, cannot throw any more exceptions than parent's version

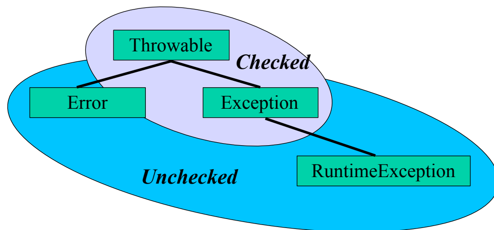
83

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

84

Exception Hierarchy



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?

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 (ArrayIndexOutOfBounds 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 */  }
```


Masking Exceptions

- Handle exception and continue

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

Reflecting Exceptions

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

```
public static int min(int[] a) {
  int m;
  try { m = a[0]; }
  catch (IndexOutOfBoundsException e) {
    throw new EmptyException();
  } … }
```

89

Exception Chaining

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

```
public static int min(int[] a) {
  int m;
  try { m = a[0]; }
  catch (IndexOutOfBoundsException e) {
    // e can be retrieved with getCause()
    throw new EmptyException("min", e);
  } … }
```

90