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
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

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
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 */
}
```

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)
Example Application

```java
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(Exception e) {
            System.out.println(e.stackTrace());
        }
    }
}
```

Creating New Exceptions

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

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

You should familiarize yourself

• Packages
  – java.lang
  – java.util
  – java.net
  – java.io

• Read the documentation on line
I/O and Network Libraries

I/O streams

- Raw communication takes place using *streams*

- Java also provides *readers* and *writers*
- 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

OutputStream - bytes

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

InputStream - bytes

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

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
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

Example Echo Application

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

Java Networking

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

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

Server code

```
server
```

Client code

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

client
Example Client/Server

**Server code**

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

**Client code**

```
Socket conn = new Socket(5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();
```
Example Client/Server

Server code

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

Note: The server is still listening other connection requests

Client code

Socket conn = new Socket (5001);
InputStream in = conn.getInputStream();
OutputStream out = conn.getOutputStream();

Client code
Example Client/Server

Server code

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

Client code

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

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
Utility Libraries

java.util

• Lots of stuff
  – Vector
  – Dictionaries
  – Enumerations and Bitsets
  – Container classes
• We will focus on Container classes
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

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
Container class hierarchy

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, no duplicates
  - List: ordered, can contain duplicate elements
    - LinkedList – unordered, dynamic size, add/delete quick
    - ArrayList – unordered, dynamic size, random access
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)

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
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
  - $\text{sgn}(\text{compareTo}(y)) = -\text{sgn}(\text{compareTo}(x))$
    - implies that $\text{compareTo}(y)$ must throw an exception iff $\text{compareTo}(x)$ throws an exception.
  - $(\text{compareTo}(y)>0 && \text{compareTo}(z)>0) \Rightarrow \text{compareTo}(z)>0$.
  - $\text{compareTo}(y)==0 \Rightarrow \text{sgn}(\text{compareTo}(z)) == \text{sgn}(\text{compareTo}(z))$
  - Recommended that $(\text{compareTo}(y)==0) == (\text{equals}(y))$

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
  - $\text{sgn}(\text{compare}(x, y)) = -\text{sgn}(\text{compare}(y, x))$
    - (This implies that compare(x, y) must throw an exception if and only if compare(y, x) throws an exception.)
  - $((\text{compare}(x, y)>0) && (\text{compare}(y, z)>0)) \Rightarrow \text{compare}(x, z)>0$.
  - $\text{compare}(x, y)==0 \Rightarrow \text{sgn}(\text{compare}(x,z))==\text{sgn}(\text{compare}(y, z))$
  - recommended $(\text{compare}(x, y)==0) == (\text{equals}(y))$
- public boolean equals(Object obj)
  - Indicates whether some other object is "equal to" this Comparator.
Example 1

```java
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]);
        }
    }
}
```

Example 2

```java
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]);
        }
    }
}
```
<table>
<thead>
<tr>
<th>MyPoint[x=1,y=95]</th>
<th>MyPoint[x=2,y=0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyPoint[x=2,y=16]</td>
<td>MyPoint[x=0,y=15]</td>
</tr>
<tr>
<td>MyPoint[x=3,y=26]</td>
<td>MyPoint[x=18,y=4]</td>
</tr>
<tr>
<td>MyPoint[x=12,y=95]</td>
<td>MyPoint[x=39,y=13]</td>
</tr>
<tr>
<td>MyPoint[x=22,y=55]</td>
<td>MyPoint[x=39,y=19]</td>
</tr>
<tr>
<td>MyPoint[x=30,y=73]</td>
<td>MyPoint[x=42,y=23]</td>
</tr>
<tr>
<td>MyPoint[x=31,y=42]</td>
<td>MyPoint[x=65,y=5]</td>
</tr>
<tr>
<td>MyPoint[x=66,y=33]</td>
<td>MyPoint[x=38,y=74]</td>
</tr>
<tr>
<td>MyPoint[x=70,y=33]</td>
<td>MyPoint[x=80,y=40]</td>
</tr>
<tr>
<td>MyPoint[x=80,y=31]</td>
<td>MyPoint[x=87,y=62]</td>
</tr>
</tbody>
</table>

Sort2 …….. // after sort
Sort3…….. // after sort