Preconditions

- Functions often have requirements on their inputs
  ```java
  // 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

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

Signaling Errors

- Style 1: Return invalid value
  ```java
  // Returns value key maps to, or null if no
  // such key in map
  Object get(Object key);
  ```
  - What if you need to say what happened?

- Style 2: Return an invalid value and status
  ```java
  static int lock_rdev(mdk_rdev_t *rdev) {
    ... 
    if (bdev == NULL) 
      return -ENOMEM; 
    ... 
  }
  ```
  ```java
  // Returns NULL if error and sets global
  // variable errno
  FILE *fopen(const char *path, const char *mode);
  ```
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

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

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

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 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"); } 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");
    }
  } 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

```java
try {
    ... a[0] ... 
} catch (IndexOutOfBoundsException e) {
    throw new EmptyException("Arrays.min");
}
```

java.lang.Throwable

• Exception is a subclass of Throwable
• 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(IOException e) {
            e.printStackTrace();
        }
    }
}
```

Creating New Exceptions

• Can make user-defined exceptions by extending Exception

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

Exception Hierarchy

```
Throwable
   - Exception
       - RuntimeException
   - Error
```

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?
Java Libraries

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

OutputStream - bytes

- Example classes
  - ByteArrayInputStream – goes to byte []
  - FileOutputStream – goes to file
- Wrappers – wrapped around OutputStream
  - BufferedReader
  - 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
  - PushBackInputStream
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.println(...)** 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;
      while((s = in.readLine()) != null) System.out.println((i++) + " "+ s);
   }
}
```

Java Networking

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

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

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

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
  - Lists
  - Maps
  - Sets
  - Iterators

Other libraries

- java.lang.Math
  - Abstract final class – only static members
  - Includes constants e and π
  - 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
- Map
- Set
- List
- SortedMap
- SortedSet

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
  - sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
    - (x.compareTo(y)>0 && y.compareTo(z)>0) => x.compareTo(z)>0.
    - sgn(x.compareTo(y)) == sgn(y.compareTo(z))
      - Recommended that (x.compareTo(y)==0) == (x.equals(y))

Comparator Interface

- When natural order isn't acceptable
- public static 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))
    - ((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.

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.print(points[i] + "n");
            System.out.println("-----------");
        }
        Arrays.sort(points); //Print the points
        for (int i=0; i<points.length; i++){
            System.out.println(points[i] + "n");
        }
    }
}
```

Output

- Sort3 ...... // after sort
  MyPoint[x=1,y=95] MyPoint[x=2,y=9]
  MyPoint[x=2,y=16] MyPoint[x=3,y=15]
  MyPoint[x=3,y=26] MyPoint[x=18,y=4]
  MyPoint[x=12,y=95] MyPoint[x=38,y=13]
  MyPoint[x=22,y=55] MyPoint[x=38,y=19]
  MyPoint[x=30,y=73] MyPoint[x=42,y=23]
  MyPoint[x=31,y=42] MyPoint[x=65,y=5]
  MyPoint[x=66,y=33] MyPoint[x=38,y=74]
  MyPoint[x=70,y=33] MyPoint[x=80,y=40]
  MyPoint[x=80,y=31] MyPoint[x=87,y=62]
- Sort2 ...... // after sort
  MyPoint[x=2,y=0] MyPoint[x=2,y=0]
  MyPoint[x=2,y=16] MyPoint[x=3,y=15]
  MyPoint[x=3,y=26] MyPoint[x=18,y=4]
  MyPoint[x=12,y=95] MyPoint[x=38,y=13]
  MyPoint[x=22,y=55] MyPoint[x=38,y=19]
  MyPoint[x=30,y=73] MyPoint[x=42,y=23]
  MyPoint[x=31,y=42] MyPoint[x=65,y=5]
  MyPoint[x=66,y=33] MyPoint[x=38,y=74]
  MyPoint[x=70,y=33] MyPoint[x=80,y=40]
  MyPoint[x=80,y=31] MyPoint[x=87,y=62]