Java Packages

- Package - is a collection of related classes and interfaces that provides access protection and namespace management.
  - Avoid name conflicts
  - Access control
- Put a **package** statement at the top of each source file in which the classes and interfaces of that package are defined

```
package mygraphics;
public class Rectangle extends Graphic
  implements Draggable {
    . . .
}
```

- To use the classes and interfaces in another package, you need to **import** the package.

Creating Packages

- If you do not use a package statement, your class or interface ends up in the *default package* - no package name
- Full class name is `mygraphics.Rectangle` in

```
package mygraphics;
public class Rectangle extends Graphic implements
  Draggable {
    . . .
}
```

  - Avoids conflict over Class Name.
- Package name:
  - reversed Internet domain name:

```
com.company.package
com.company.region.package
ibm.watson.graphics
```

Using Packages

- Outside access to package classes which are public -either:
  - By long name:

```
mygraphics.Rectangle myRect = new mygraphics.Rectangle();
```

  - Import specific class:

```
package  currentpkgname ;
import mygraphics.Rectangle
Rectangle myRect = new Rectangle();
```

  - To import *all* of the classes and interfaces of particular package:

```
import graphics.*;
```

- Java runtime system automatically imports three packages:
  - default package (the package with no name)
  - **`java.lang package`**
  - The current package

Class Path

- *Class path* - is a list of directories or zip files to search for class files - used by both the compiler and the interpreter.
- To change your class path:
  - Set the `CLASSPATH` environment variable *(not recommended).*
  - Use the `-classpath` runtime option when you invoke the compiler or the interpreter.
- DOS shell (Windows 95/NT):

```
javac -classpath .;C:\classes;C:\JDK\lib\classes.zip
```

- UNIX:

```
javac -classpath .:~/classes:/JDK/lib/classes.zip
```

- **-classpath** completely overrides current class path:
  - Must include the **classes.zip** file from the JDK in the class path. The current directory is good idea too.

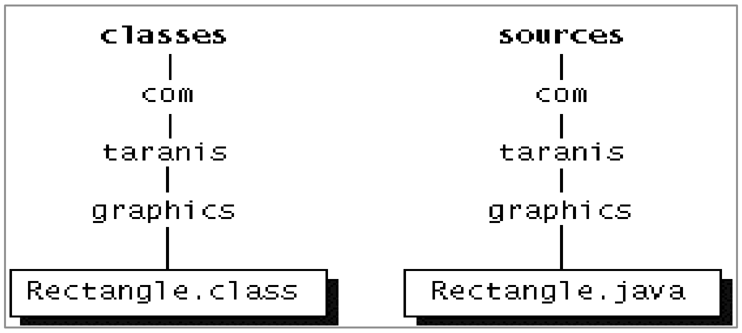
Naming Packages

- Hierarchical naming and file system:
  - file name = short name of class or interface .java
  - file path is full name of package
  - Example: class RateRectangle in East division of company MacroMicro @ East.MacroMicro.com should be in file

    $classpath /com/MacroMicro/East/RateRectangle.java

  with package name:

  package com.MacroMicro.East.RateRectangle



Exception Handling

- An **exception** is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
  - E.g.: trying to access an out-of-bounds array element.
  - *throwing an exception* creates an **exception object**

```
InputFile.java:11: Exception java.io.FileNotFoundException
must be caught, or it must be declared in the throws clause
of this method.
        in = new FileReader(filename);
             ^
```

- Java language requires that a method either *catch* all "checked" exceptions (those that are checked by the runtime system) or specify that it *throws* that type of exception.

Separating Error Handling Code from "Regular" Code

- A function that reads an entire file into memory.
  In **pseudo-code**, your function might look something like this:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

- It ignores all of these potential errors:
  - What happens if the file can't be *opened*?
  - What happens if the *length* of the file can't be determined?
  - What happens if enough *memory* can't be allocated?
  - What happens if the *read* fails?
  - What happens if the file can't be *closed*?

In-line error checking - with*out* Java Exceptions

Original **7** lines (in italics) inflated to 29 lines of code--**bloat** factor of **400** %

The logical flow of the code has been lost in the clutter

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpened) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose &&
            errorCode == 0) {
                errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

With Java Exception Handling

Separating Error Handling Code from "Regular" Code

```
readFile {
    try {
        open the file;
         determine its size;
         allocate that much memory;
         read the file into memory;
         close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Another Example: Handling Errors at each level w / **out** Exceptions

```
method1 {
     call method2;
}
method2 {
     call method3;
}
method3 {
     call readFile;
}
```

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}
errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}
errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

Propagating Errors Up the Call Stack
with Java Exceptions

```
method1 {
    try {
        call method2;
    } catch (exception) {
        doErrorProcessing;
    }
}
method2 throws exception {
    call method3;
}
method3 throws exception {
    call readFile;
}
```

Exception Hierarchy

- **All exceptions within a Java program are first-class objects**
- Exception Classes form a hierarchy. Each intermediate class node represents a group of related exception types.
  - The lowest level provides the most selectivity.

```
java.lang.Object
  |
  +--java.lang.Throwable
       |
       +--java.lang.Exception
            |
            +--java.io.IOException
                 |
                 +-- java.io.FileNotFoundException
```

Using the Exception Hierarchy

- Catch *individual* type of exception:
  - catch (InvalidIndexException e) { . . . }
- Catch a *group* of exceptions:
  - catch (ArrayException e) { . . . }
- Catch *all* exceptions:
  - catch (Exception e) { . . . }
  - makes your code more *error prone* by catching and handling exceptions that you didn't anticipate and therefore are *not* correctly handled within the handler

Creating an Exception Handler

- Enclose the statements that might throw an exception within a `try block`:

```
try {
     Java statements
}
```

- Catch blocks *directly after* the try block:

```
try {
     . . .
} catch ( . . . ) {
     . . .
} catch ( . . . ) {
     . . .
} . . .
```

- General form of Java's catch statement is:

```
catch (SomeThrowableObject variableName) {
     Java statements
}
```

Creating a *Specific* Exception Handler

- ***`SomeThrowableObject`*** **must be subclass of Java.lang.Throwable**
-
```
try {
      . . .
 } catch (ArrayIndexOutOfBoundsException e) {
     System.err.println("Caught ArrayIndexOutOfBoundsException: " +
            e.getMessage());
 } catch (IOException e) {
     System.err.println("Caught IOException: " +
            e.getMessage());
 }
```

Creating a *General* Exception Handler

- An exception handler that handles both types of exceptions:

```
try {
    . . .
} catch (Exception e) {
    System.err.println("Exception caught: " +
 e.getMessage());
}
```

- Exception handlers should be *more specialized*.
  - General Exception handlers are too error prone, and more difficult to debug.
- Code within a **`finally`** block
  - will be executed regardless of whether control exits the **`try`** block due to an exception scenario or normally.

IO Exception Hierarchy

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--java.lang.RuntimeException
                    |
                    +--java.lang.IndexOutOfBoundsException
                           |
                           +--java.lang.ArrayIndexOutOfBoundsException
                           |
                           +--java.lang.StringIndexOutOfBoundsException
```

Throws vs Throw - *very different*

- **`throws` - pass-along the exception :**

```
public void writeList() throws IOException,
                        ArrayIndexOutOfBoundsException {
```

  - ArrayIndexOutofBoundsException is a *runtime* exception, so you don't have to specify it in the throws clause, tho you can.

- **`throw` - create an exception object :**
  - `throw` requires single argument: *a throwable object*.
    - An instance of any subclass of the **Throwable** class.

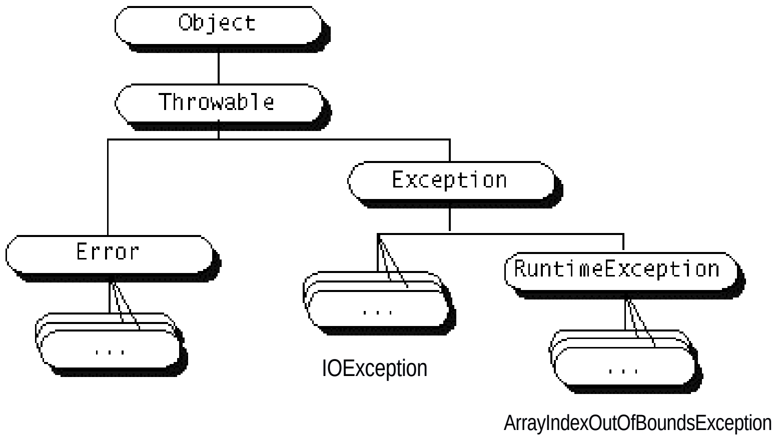
    `throw someThrowableObject;`

```
// Note: This class won't compile by design!
import java.io.*;
import java.util.Vector;

public class ListOfNumbers {
    private Vector victor;
    private static final int size = 10;

    public ListOfNumbers () {
        victor = new Vector(size);
        for (int i = 0; i < size; i++)
            victor.addElement(new Integer(i));
    }
    public void writeList() {
! ➡      PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

        for (int i = 0; i < size; i++)
➡          out.println("Value at: " + i + " = " + victor.elementAt(i));

        out.close();
    }
}
```

Completed Try / Catch / Finally

```
public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entering try statement");
        out = new PrintWriter(
                  new FileWriter("OutFile.txt"));

        for (int i = 0; i < size; i++)
            out.println("Value at: " + i + " = " +
                                  victor.elementAt(i));
    } catch (ArrayIndexOutOfBoundsException e) {
      System.err.println("Caught ArrayIndexOutOfBoundsException: "
                           + e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: "
                                        + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        } else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

Exception Class Hierarchy


Object
Throwable
Exception
Error
RuntimeException
. . .
. . .
. . .
IOException
ArrayIndexOutOfBoundsException