Interfaces and Packages

What is an Interface?

- An *interface* defines a protocol of behavior that can be implemented by any class anywhere in the class hierarchy.
- An interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior.
- An *interface* is a named collection of method definitions (without implementations).
- Interface reserve behaviors for classes that implement them.

- Methods declared in an interface are always public and abstract, therefore Java compiler will not complain if you omit both keywords
- Static methods cannot be declared in the interfaces – these methods are never abstract and do not express behavior of objects
- Variables can be declared in the interfaces. They can only be declared as static and final. – Both keyword are assumed by default and can be safely omitted.
- Sometimes interfaces declare only constants – be used to effectively import sets of related constants.

Interface vs. Abstract Class

- An interface is simply a list of unimplemented, and therefore abstract, methods.
- An interface cannot implement any methods, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

Defining an Interface

- Defining an interface is similar to creating a new class.
- An interface definition has two components: the interface declaration and the interface body.

  interfaceDeclaration

  {

  interfaceBody

  }

– The *interfaceDeclaration* declares various attributes about the interface such as its name and whether it extends another interface.

– The *interfaceBody* contains the constant and method declarations within the interface

```
public interface StockWatcher
{
    final String sunTicker = "SUNW";
    final String oracleTicker = "ORCL";
    final String ciscoTicker = "CSCO";
    void valueChanged
     (String tickerSymbol,
     double newValue);
}
```

If you do not specify that your interface is public, your interface will be accessible only to classes that are defined in the same package as the interface

Implementing an Interface

- Include an implements clause in the class declaration.
- A class can implement more than one interface (the Java platform supports multiple inheritance for interfaces), so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.
- When implement an interface, either the class must implement all the methods declared in the interface and its superinterfaces, or the class must be declared abstract

```
class C
{
    public static final int A = 1;
}
interface I
{
    public int A = 2;
}
class X implements I
{
    …
    public static void main (String[] args)
    {
     int I = C.A, j = A;
     …
    }
}
```

```
public class StockMonitor
    implements StockWatcher
{
   ...
   public void valueChanged
    (String tickerSymbol, double newValue)
   {
    if (tickerSymbol.equals(sunTicker))
    { ... }
    else if
    (tickerSymbol.equals(oracleTicker))
    { ... }
    else if
   (tickerSymbol.equals(ciscoTicker))
    { ... }
   }
}
```

Properties of Interface

- A new interface is a new reference data type.
- Interfaces are not instantiated with *new,* but they have certain properties similar to ordinary classes
- You can declare that an object variable will be of that interface type

e.g.

```
Comparable x = new Tile(…);
Tile y = new Tile(…);
if (x.compareTo(y) < 0) …
```

Superinterface (1)

- An interface can extend other interfaces, just as a class can extend or subclass another class.
- An interface can extend any number of interfaces.
- The list of superinterfaces is a comma-separated list of all the interfaces extended by the new interface

  public interfaceName
   Extends superInterfaces
  {
   InterfaceBody
  }

Superinterface (2)

- Two ways of extending interfaces:
  - Add new methods
  - Define new constants
- Interfaces do not have a single top-level interface. (Like Object class for classes)

Interfaces Cannot Grow!

- Add some functionality to StockWatcher?

```
public interface StockWatcher
{
  final String sunTicker = "SUNW";
  final String oracleTicker = "ORCL";          final
String ciscoTicker = "CSCO";
  void valueChanged(String tickerSymbol,
        double newValue);
  void currentValue(String tickerSymbol,
        double newValue);
}
```

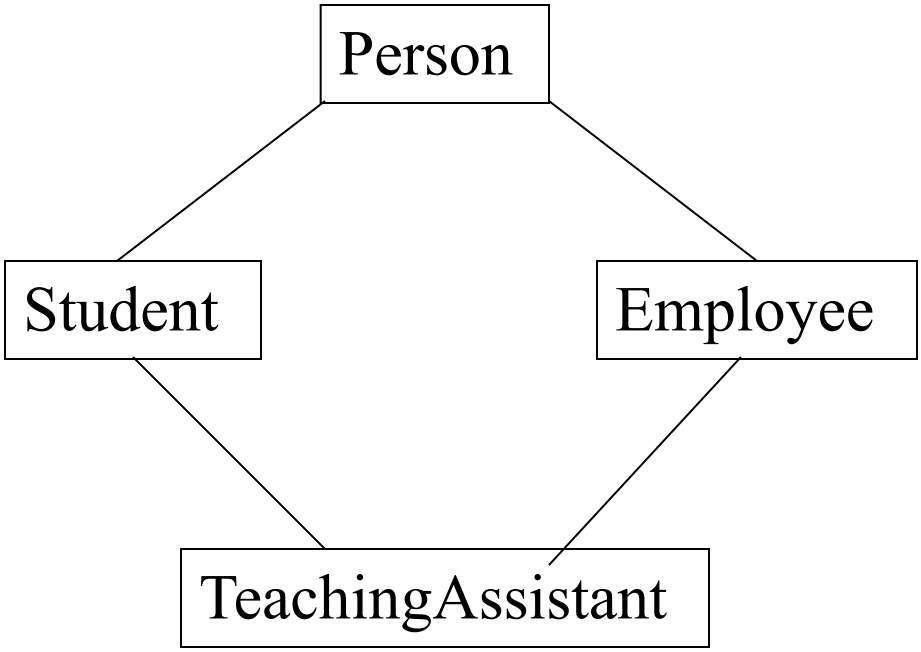
- If you make this change, all classes that implement the old interface will break because they don't implement the interface anymore!

- Try to anticipate all uses for your interface up front and specify it completely from the beginning.
- Create a StockWatcher subinterface called StockTracker that declared the new method:

```
public interface StockTracker
  extends StockWatcher
{
  void currentValue(String tickerSymbol,
        double newValue);
}
```

- Users can choose to upgrade to the new interface or to stick with the old interface.

Multiple Inheritance



- Multiple inheritance of class is not allowed in Java

```
class TeachingAssistant extends
    Student
{
    private EmployeeAttributes ea;
    …
    public String getEmployeeID()
    {
     return this.ea.getEmployeeID();
    }
}
```

EmployeeAttributes – non-public utility class

```
Interface Comparable
{
   int compareTo(Object o);
}

Class Student extends Person
     implements comparable
{
   parivate int SN; //Student number
   …
   public int compareTo(Object o)
   {
    return this.SN – ((Student)o).SN;
   }
   …
}
```

```
interface x
{
 char A = ‘A’;
 void gogi();
}
interface Y
{
 char B = ‘B’;
}
interface Z extends X, Y
{
 void dogi();
}
```

?What is in interface Z

Name conflicts in extending interfaces

- A class automatically implements all interfaces that re implemented by its superclass
- Interfaces belong to Java namespace and as such are placed into packages just like classes.
- Some interfaces are declared with entirely empty bodies. They serve as labels for classes
  - The most common marker interfaces are Cloneable and Serializable

```
class Car implement Cloeable
{ …
  public Object clone()
  {     return super.clone();
  }
}
```

```
interface X
{   char A = ‘A’;
    void gogi();
    void dogi();
}
interface Y
{   char A = ‘B’;
    void gogi(); //but not int gogi()
}
interface Z extends X, Y
{   void dogi(int i);
}
class C implements Z
{   public void gogi()
    {       char c = Y.A; …
    }
    public void dogi()
    {       char c = X.A; …
    }
    public void dogi(int I)
    {       …
     this.dogi(i – 1);
    }
}
```

Interface and Callbacks

- Suppose you want to implement a Timer class. You want you program to be able to:
  - Stat the timer;
  - Have the timer measure some time delay
  - Then carry out some action when the correct time has elapsed.
- The Timer needs a way to communicate with the calling class – callback function

```
class Timer extends Thread
{
    …
    public void run()
    {
        while (true)
        {
            sleep(delay);
            //now what?
        }
    }
}
```

The object constructing a Timer object must somehow tell the timer what to do when the time is up.

```
interface TimerListener
{
    public void timeElapsed(Timer t);
}
```

```
class Timer extends Thread
{
    Timer(TimerListener t) { listener = t;}
    …
    public void run()
    {       while (true)
     {
            sleep(interval);
            listener.timeElapsed(this);
     }
     TimerListener listener;
    }
}
class AlarmClock implements TimerListener
{   AlarmClock()
    {
     Timer t = new Timer(this);
     t.setDelay(1000); //1000 milliseconds
    }
    public void timeElapsed(Timer t)
    {
     if (t.getTime() >= wakeUpTime) wakeUp.play();
    }
}
```

What is Package?

- A *package* is a collection of related classes and interfaces providing access protection and namespace management.
  - To make classes easier to find and to use
  - To avoid naming conflicts
  - To control access

Why Using Packages?

- Programmers can easily determine that these classes and interfaces are related.
- Programmers know where to find classes and interfaces that provide graphics-related functions.
- The names of classes won't conflict with class names in other packages, because the package creates a new namespace.
- Allow classes within the package to have unrestricted access to one another yet still restrict access for classes outside the package

Put Your Classes and Interfaces into Packages

- It is no need to “create” packages, they come to existence as soon as they are declared
- The first line of your source file:

  package <package_name>;

  e.g. package cars;

  class Car

  { … }
- Every class must belong to one package.
- Only one package statement is allowed per source file.
- If package statement is omitted, the class belongs to a special default package – the only package in Java that has no name.

Subpackages

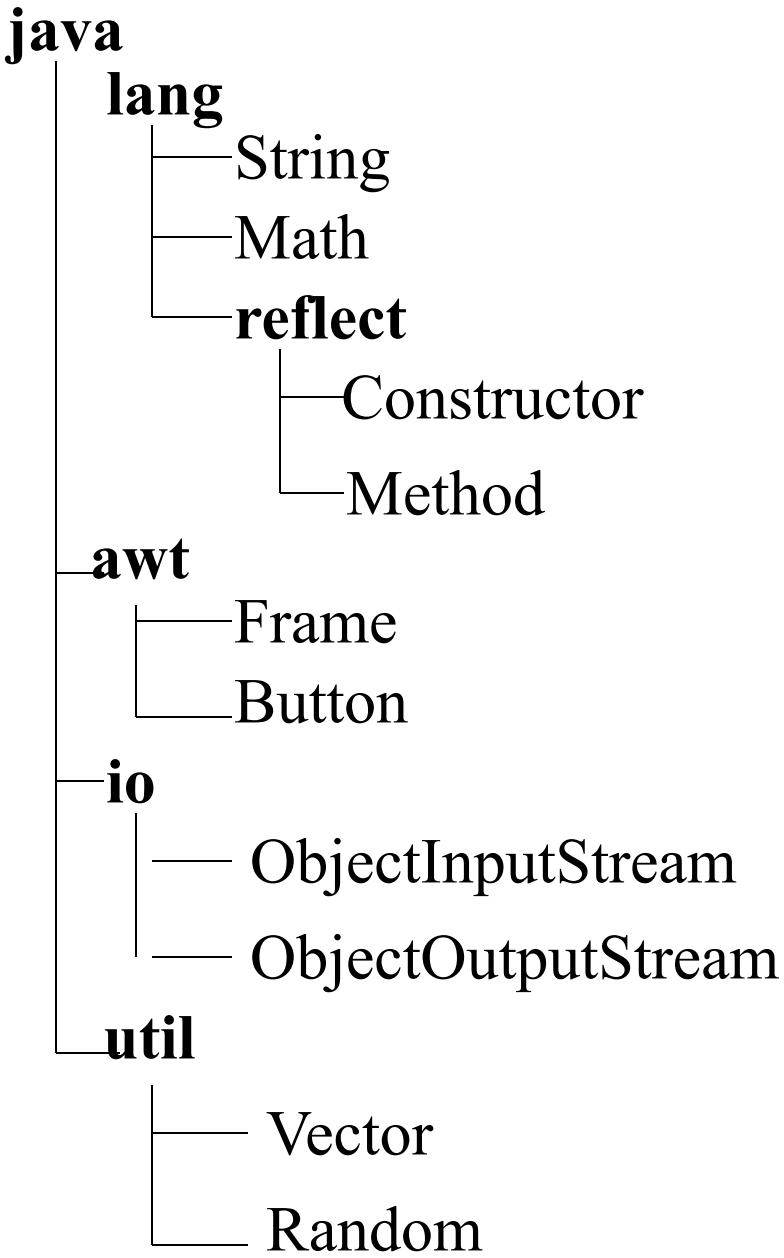
- We can create hierarchies of nested packages

  e.g.

  package machines.vehicles.cars;
  class FastCar extends Car
  {…}

- Subpackage names must always be prefixed with names of all enclosing packages.
- Package hierarchy does not have a single top-level all-encompassing root package.Instead, we deal with a collection of top-level packages

Partial Package Tree of Java



Packages and Class

- Packages subdivide name space of Java classes

```
machines.vehicles.cars.FastCar mycar =
        new machine.vehicles.cars.FastCar();
```

- Package names should be made as unique as possible to prevent name clashes
- Class name must be fully qualified with their package name. Except:
  - Class name immediately following the class keyword, e.g. class Car{…}
  - Class belongs to the same package as the class being currently defined
  - Class is explicitly specified in one of the import statements
  - Class belongs to java.lang package (all java.lang classes are implicitly imported)

import Statement

- To avoid typing fully qualified names – use import statements sandwiched between the package statement and class definition.

```
e.g.  package grand.prix;
      import java.lang.*;     //implicitly specified
      import java.util.Random;
      import machines.vehicles.cars.*;
      import machines.*.*; //COMPILER ERROR!
      import Racer;     //from default package

      class SuperFastCar extends FastCar
      {
        private Racer r;
        public static void main(String{[] args)
        {
              Random RNG = new Random();
              java.util.Vector v =
                    new java.util.Vector();
        }
      }
```

Store Class Files

- Class files must be stored in a directory structure that mirrors package hierarchy
  - e.g. Both .java and .class files for FastCar class from the machines.vehicles.cars package can be stored in the following directory:

C:\A\B\classes\machines\vehicles\cars

Classpath

- Compiled classes can be stored in different locations in the file systems.
- How to locating these files – Setting a classpath which is a list of directories where class files should be searched

  e.g. If Java system is to find classes from the machines.vehicles.cars package , its classpath must point to C:\A\B\classes, the root of the package directory hierarchy.

Setup Classpath

- In command line

  c:\> java –classpath C:\A\B\classes machines.vehicles.cars.Car

- To avoid specify the classpath in every command, set classpath as a command shell environment variable:
  - Windows:

  >set CLASSPATH = C:\A\B\classes
  >set CLASSPATH = .; %CLASSPATH%
  >echo %CLASSPATH%
  .; C:\A\B\classes

  - Unix:

  $CLASSPATH = $HOME/A/B/classes
  $CLASSPATH = .: $CLASSPATH
  $echo $CLASSPATH
  .: home/santa/A/B/classes

Classpath for .zip and .jar File

- JDK can open .zip and .jar archives and search for class files inside.
- The archives must store class files within the appropriate directory structure.
- Set up classpath for archives:

  >set CLASSPATH = %CLASSPATH%;c:\user.jar

Exercise

1. What methods would a class that implements the java.util.Iterator interface have to implement?
2. What is wrong with the following interface?

```
public interface SomethingIsWrong {
    public void aMethod(int aValue)
    {
        System.out.println("Hi Mom");
    }
}
```

3. Fix the interface in question 2.
4. Is the following interface valid?

   public interface Marker { }

5. Write a class that implements the Iterator interface found in the java.util package. The ordered data for this exercise is the 13 cards in a suit from a deck of cards. The first call to next returns 2, the subsequent call returns the next highest card, 3, and so on, up to Ace. Write a small main method to test your class.
6. Suppose that you have written a time server, which periodically notifies its clients of the current date and time. Write an interface that the server could use to enforce a particular protocol on its clients.

Exception and Error Handling

What's an Exception?

- *exception* is shorthand for the phrase "exceptional event."
- An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
- Many kinds of errors can cause exceptions
  - Hardware errors, like a hard disk crash,
  - Programming errors, such as trying to access an out-of-bounds array element.

Process in Java When Errors Happen

- When an error occurs within a Java method, the method creates an exception object and hands it off to the runtime system.
  - The exception object contains information about the exception, including its type and the state of the program when the error occurred.
- The runtime system is responsible for finding some code to handle the error.
- Creating an exception object and handing it to the runtime system is called *throwing an exception*.

Why Using Exceptions?

- Java programs have the following advantages over traditional error management techniques:
  - Separate error handling code from “regular” code.
  - Propagate errors up the call stack
  - Group error types and error differentiation

Separating Error Handling Code from "Regular" Code

- In traditional programming, error detection, reporting and handling often lead to confusing code.
  - A high rate of a bloat factor
  - So much error detection, reporting and returning that the original code are lost in the clutter.
  - The logical flow of the code has been lost in the clutter, making it difficult to tell if the code is doing the right thing:
  - It's even more difficult to ensure that the code continues to do the right thing after you modify the function three months after writing it.
  - Ignoring it?--errors are "reported" when their programs crash.

e.g. The pseudo-code for a function that reads an entire file into memory might look 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?

```
errorCodeType readFile
{   initialize errorCode = 0;
    open the file;
    if (theFileIsOpen)
    {       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;
}
```

- **Java solution:**

```
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; }
}
```

Propagating Errors Up the Call Stack

- Propagate error reporting up the call stack of methods.
  - Suppose that the readFile method is the fourth method in a series of nested method calls: method1 calls method2, which calls method3, which finally calls readFile.

    method1 { call method2; }

    method2 { call method3; }

    method3 { call readFile; }

    Suppose that method1 is the only method interested in the errors that occur within readFile.
  - Traditional error notification techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1--the only method that is interested in them.

```
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;
}
```

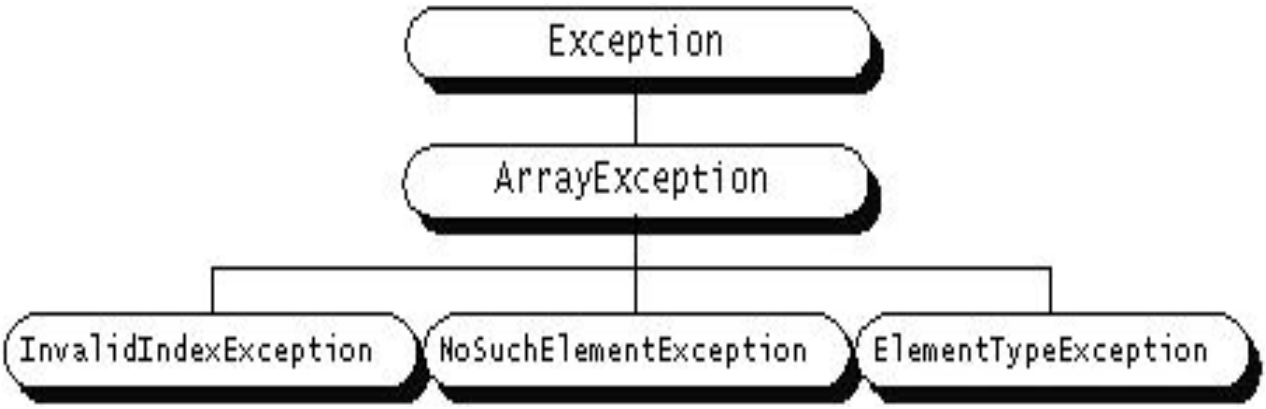
- Java solution: runtime system searches backwards through the call stack to find any methods that are interested in handling a particular exception.
- A Java method can "duck" any exceptions thrown within it, thereby allowing a method further up the call stack to catch it. Thus only the methods that care about errors have to worry about detecting errors.

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

Grouping Error Types and Error Differentiation

- Exceptions often fall into categories or groups.
  - A group of exceptions:
    - The index is out of range for the size of the array
    - The element being inserted into the array is of the wrong type
    - The element being searched for is not in the array.
  - Some methods would like to handle all exceptions that fall within a category, and other methods would like to handle specific exceptions
- All exceptions that within a Java program are first-class objects, grouping or categorization of exceptions is a natural outcome of the class hierarchy.

- Java exceptions must be instances of Throwable or any Throwable descendant.
- You can create subclasses of the Throwable class and subclasses of your subclasses.
- Each "leaf" class (a class with no subclasses) represents a specific type of exception and each "node" class (a class with one or more subclasses) represents a group of related exceptions



Overview of Exception Hierarchy (1)

- When an exceptional condition causes an exception to be *thrown*, that exception is an object derived, either directly, or indirectly from the class **Throwable**
- The Throwable class has two subclasses:
  - Error: indicates that a non-recoverable error has occurred that should not be caught. Errors usually cause the Java interpreter to display a message and exit
  - Exception: indicates an abnormal condition that must be properly handled to prevent program termination

Overview of Exception Hierarchy (2)

- In JDK 1.1.3, there are nine subclasses of the Exception class, several of which have numerous subclasses.
  - One subclass of Exception is the class named RuntimeException This class has eleven subclasses, some of which are further subclasses.
- All exceptions other than those in the RuntimeException class must be either
  - caught
  - declared (specified) in a *throws* clause of any method that can throw them.

Throwable Class Definition

```
public class java.lang.Throwable
      extends java.lang.Object
{
    // Constructors
    public Throwable();
    public Throwable(String message);

    // Methods
    public Throwable fillInStackTrace();
    public String getMessage();
    public void printStackTrace();
    public void printStackTrace(PrintStream s);
    public String toString();
}
```

Runtime Exceptions

- Exceptions that occur within the Java runtime system.
  - arithmetic exceptions (e.g. dividing by zero)
  - pointer exceptions (e.g. trying to access an object through a null reference)
  - indexing exceptions (e.g. attempting to access an array element through an index that is too large or too small).
- Runtime exceptions can occur anywhere in a program.
- The cost of checking for runtime exceptions often exceeds the benefit of catching or specifying them. Thus the compiler does not require catching or specifying runtime exceptions.

Some Terms in Exception Handling (1)

- Catch: A method *catches* an exception by providing an exception handler for that type of exception object.
- Specify: If a method does not provide an exception handler for the type of exception object thrown, the method must *specify* (declare) that it can throw that exception.
  - Any exception that can be thrown by a method is part of the method's public programming interface and users of a method must know about the exceptions that a method can throw.
  - You must specify the exceptions that the method can throw in the method signature

Some Terms in Exception Handling (2)

- **Checked Exceptions:** exceptions that are not runtime exceptions and are checked by the compiler
  - Java has different types of exceptions, including I/O Exceptions, runtime exceptions and exceptions of your own creation, etc.
  - Checked exceptions are exceptions that are not runtime exceptions.
  - Exceptions of all Exception classes and subclasses other than Runtime Exception which is a subclass of Exception are checked by the compiler and will result in compiler errors if they are not either *caught* or *specified*

Some Terms in Exception Handling (3)

- Exceptions That Can Be Thrown Within the Scope of the Method
  - The throw statement includes more than just the exceptions that can be thrown directly by the method:
  - This phrase includes any exception that can be thrown while the flow of control remains within the method. This statement includes both
    - Exceptions that are thrown directly by the method with Java's throw statement.
    - Exceptions that are thrown indirectly by the method through calls to other methods

```
import java.io.*;
public class InputFile
{
    private FileReader in;
    public InputFile(String filename)
    {
     in = new FileReader(filename);
    }
    public String getWord()
    {
     int c;
     StringBuffer buf = new StringBuffer();
     do
     {
          c = in.read();
          if (Character.isWhitespace((char)c))
               return buf.toString();
          else buf.append((char)c);
     } while (c != -1);
     return buf.toString();
    }
}
```

Compiler give error message about the bold line. - The FileReader read method throws a java.io.IOException if for some reason it can't read from the file

```
import java.io.*;

public class InputFileDeclared
{
    private FileReader in;
    public InputFileDeclared(String filename)
     throws FileNotFoundException
    {
     in = new FileReader(filename);
    }
    public String getWord() throws IOException
    {       int c;
     StringBuffer buf = new StringBuffer();
     do
     {
           c = in.read();
           if (Character.isWhitespace((char)c))
                 return buf.toString();
           else buf.append((char)c);
     } while (c != -1);
     return buf.toString();
    }
}
```

The try Block (1)

- The first step in writing any exception handler is putting the Java statements within which an exception can occur into a try block.
- The try block is said to govern the statements enclosed within it and defines the scope of any exception handlers (established by subsequent catch blocks) associated with it.
- Syntax:

```
try
{
 //java statements
}
```

The try Block (2)

- Put statements in *try* block
  - Put each statement that might throw exceptions within its own *try* block and provide separate exception handlers for each *try* block
    - Some statements, particularly those that invoke other methods, could potentially throw many types of exceptions.
    - A *try* block consisting of a single statement might require many different exception handlers.
  - Put all of the statements that might throw exceptions within a single *try* block and associate multiple exception handlers with it.

The try Block (3)

- Exception handlers must be placed immediately following their associated *try* block.
- If an exception occurs within the *try* block, that exception is handled by the appropriate exception handler associated with the *try* block.

The catch Block(s) (1)

- Associate exception handlers with a try block by providing one or more catch blocks directly after the try block
- The catch block contains a series of legal Java statements.
  - These statements are executed if and when the exception handler is invoked.
- Syntax:

```
catch (AThrowableObjectType variableName)
{
  //Java statements
}
```

The catch Block(s) (2)

- The runtime system invokes the exception handler when the handler is the first one in the call stack whose type matches that of the exception thrown
  - The order of your exception handlers is important, particularly if you have some handlers which are further up the exception hierarchy tree than others
  - Those handlers designed to handle exceptions furthermost from the root of the hierarchy tree should be placed first in the list of exception handlers
    - Otherwise, those handler designed to handle a specialized "leaf" object may be preempted by another handler whose exception object type is closer to the root if the second exception handler appears earlier in the list of exception handlers

Argument of the catch Block(s)

- The argument type declares the type of exception object that a particular exception handler can handle and must be the name of a class that inherits from the Throwable class.
- As in a method declaration, there is a parameter which is the name by which the handler can refer to the exception object.
  - You access the instance variables and methods of exception objects the same way that you access the instance variables and methods of other objects

Catching Multiple Exception Types with One Handler

- **Java allows to write general exception handlers that handle multiple types of exceptions**
  - **Exception handler can be written to handle any class that inherits from Throwable.**
  - **If you write a handler for a "leaf" class (a class with no subclasses), you've written a specialized handler: it will only handle exceptions of that specific type.**
  - **If you write a handler for a "node" class (a class with subclasses), you've written a general handler: it will handle any exception whose type is the node class or any of its subclasses**

Catch These Exceptions

- To catch only those that are instances of a leaf class.
  - An exception handler that handles only invalid index exceptions :

    catch (InvalidIndexException e) { . . . }
- A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement.
  - To catch all array exceptions regardless of their specific type, an exception handler:

    catch (ArrayException e) { . . . }

  This handler would catch all array exceptions including InvalidIndexException, ElementTypeException, and NoSuchElementException.
- Set up an exception handler that handles any Exception :

  catch (Exception e) { . . . }

The finally Block

- *finally* block provides a mechanism that allows your method to clean up after itself regardless of what happens within the *try* block.
- The final step in setting up an exception handler is providing a mechanism for cleaning up the state of the method before allowing control to be passed to a different part of the program.
  - Do this by enclosing the cleanup code within a finally block.
  - The runtime system always executes the statements within the finally block regardless of what happens within the try block.

Specifying (Declaring) the Exceptions Thrown by a Method

- Sometimes it is best to handle exceptions in the method where they are detected, and sometimes it is better to pass them up the call stack and let another method handle them.
- In order to pass exceptions up the call stack, you must *specify* or *declare* them
  - To specify that a method throws exceptions, add a *throws* clause to the method signature for the method.

```
void myMethod()
    throws InterruptedException,
    MyException, HerException,
    UrException
{
    //method code
}//end myMethod()
```

- Any method calling this method would be required to either handle these exception types, or continue passing them up the call stack.
- Eventually, some method must handle them or the program won't compile.

The throw Statement

- Before you can catch an exception, some Java code somewhere must throw one.
- Any Java code can throw an exception: your code, code from a package written by someone else or the Java runtime system.
- Exceptions are always thrown with the Java *throw* statement
- The object be thrown may be an instance of any subclass of the Throwable class
- Syntax:

```
throw myThrowableObject;
```

Use Exception Classes

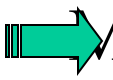
- Use a class written by someone else. For example, the Java development environment provides a lot of exception classes that you could use.
- Write an exception class of your own
  - It is possible to define your own exception classes, and to cause objects of those classes to be thrown whenever an exception occurs according to your definition of an exception.

Should I Write My Own Exception Classes?

- Do you need an exception type that isn't represented by those in the Java development environment?
- Would it help your users if they could differentiate your exceptions from those thrown by classes written by other vendors?
- Does your code throw more than one related exception?
- If you use someone else's exceptions, will your users have access to those exceptions?
- Should your package be independent and self-contained?

Choosing a Superclass

- Your exception class has to be a subclass of Throwable
- The two subclasses of Throwable: Exception and Error - live within the two existing branches
- Errors are reserved for serious hard errors that occur deep in the system: It is not likely that your exceptions would fit the Error category.



Make your new classes direct or indirect descendants of Exception

  - Decide how far down the Exception tree you want to go

```
/*Copyright 1997, R. G. Baldwin
    Illustrates creating, throwing, catching, and
    processing a custom exception object that contains
    diagnostic information.
    Also illustrates that the code in the finally block
    executes despite the fact that the exception handler
    tries to terminate the program by executing a return
    statement.
*/
//The following class is used to construct a customized
// exception object. The instance variable in the object
// is used to simulate passing diagnostic information
// from the point where the exception is thrown to the
// exception handler.

class MyPrivateException extends Exception
{
    int diagnosticData;

    //constructor
    MyPrivateException(int diagnosticInfo)
    {   //save diagnosticInfo in the object
        diagnosticData = diagnosticInfo;
    }
```

```
    //Overrides Throwable's getMessage() method
    public String getMessage()
    {
        return ("The message is: buy low, sell high\n"
        + "The diagnosticData value is: "
        + diagnosticData);
    }
}
class Excep //controlling class
{
    public static void main(String[] args)
    {
     try
     {
        for(int cnt = 0; cnt < 5; cnt++)
        {    //Throw a custom exception, and pass
             //diagnosticInfo if cnt == 3
             if(cnt == 3)
                 throw new MyPrivateException(3);
             //Transfer control before
             // "processing" for cnt == 3
```

```
                System.out.println( "Processing data”
                    + “ for cnt =:“ + cnt);
            }//end for-loop
            System.out.println( "This line of code will”
                + “ never execute.");
        }
        catch(MyPrivateException e)
        {
            System.out.println( "In exception handler, get”
                + “ the message\n" + e.getMessage());
            System.out.println( "In exception handler,”
                + “ trying to terminate program.\n”
                + “by executing a return statement");
            return; //try to terminate the program
        }
        finally
        {
            System.out.println( "In finally block just to prove”
                + “that we get here despite\n the return “
                + “statement in the exception handler.");
        }
        System.out.println( "This statement will never execute due to”
            + “ return statement in the exception handler.");
    }
}
```

Exercise

1. List five keywords that are used for exception-handling
2. All exceptions in Java are thrown by code that you write: True or False? If false, explain why.
3. Explain why you should place exception handlers furthermost from the root of the exception hierarchy tree first in the list of exception handlers
4. Provide a code fragment that illustrates how you would specify that a method throws more than one exception

5. Write a Java application that creates and then deals with an ArithmeticException caused by attempting to divide by zero.

The output from the program should be similar to the following:

Program is running. The quotient is: 3

Program is running. The quotient is: 6

Oops, caught an exception with the message: / by zero and with the stacktrace showing: java.lang.ArithmeticException: / by zero at ExcpHandle.main(ExcpHandle.java: 17)

Converting the exception object to a String we get: java.lang.ArithmeticException: / by zero In a real program, we might take corrective action here.

Solve Common Coding Problems

- **Problem** 1: The compiler complains that it can't find a class
  - Make sure you've imported the class or its package.
  - Unset the CLASSPATH environment variable, if it's set.
  - Make sure you're spelling the class name exactly the same way as it is declared. Case matters!
  - If your classes are in packages, make sure that they appear in the correct subdirectory.
  - Also, some programmers use different names for the class name from the .java filename. Make sure you're using the class name and not the filename. In fact, make the names the same and you won't run into this problem for this reason.

- **Problem 2**: The interpreter says it can't find one of my classes.
  - Make sure you type in the command:

    java myclass

    not java myclass.java
  - Make sure you specified the class name--not the class file name--to the interpreter.
  - Unset the CLASSPATH environment variable, if it's set.
  - If your classes are in packages, make sure that they appear in the correct subdirectory.
  - Make sure you're invoking the interpreter from the directory in which the .class file is located

- **Problem 3**: My program doesn't work! What's wrong with it? --The following is a list of common programming mistakes by novice Java programmers. Make sure one of these isn't what's holding you back
  - Did you forget to use `break` after each `case` statement in a `switch` statement?
  - Did you use the assignment operator = when you really wanted to use the comparison operator ==?
  - Are the termination conditions on your loops correct? Make sure you're not terminating loops one iteration too early or too late. That is, make sure you are using < or <= and > or >= as appropriate for your situation.

- Remember that array indices begin at 0, so iterating over an array looks like this:

  ```
  for (int i = 0; i < array.length; i++) . . .
  ```

- Are you comparing floating-point numbers using ==? Remember that floats are approximations of the real thing. The greater than and less than (> and <) operators are more appropriate when conditional logic is performed on floating-point numbers.
- Are you using the correct conditional operator? Make sure you understand && and || and are using them appropriately.

- Make sure that blocks of statements are enclosed in curly brackets { }. The following code looks right because of indentation, but it doesn't do what the indents imply because the brackets are missing:

```
for (int i = 0; i < arrayOfInts.length; i++)
  arrayOfInts[i] = i;
  System.out.println("[i] = " + arrayOfInts[i]);
```

- Do you use the negation operator ! a lot? Try to express conditions without it. Doing so is less confusing and error-prone.
- Are you using a do-while? If so, do you know that a do-while executes at least once, but a similar while loop may not be executed at all?

- Are you trying to change the value of an argument from a method? Arguments in Java are passed by value and can't be changed in a method.
- Did you inadvertently add an extra semicolon (;), thereby terminating a statement prematurely? Notice the extra semicolon at the end of this for statement:

```
for (int i = 0; i < arrayOfInts.length; i++) ;
  arrayOfInts[i] = i;
```

Useful Tricks for Debugging

1. Print the value of any variable with code like this:

```
System.out.println("x = " + x);
```

2. To get the state of current object, print the state of this object:

```
System.out.println("Entering loadImage. This = " +
        this);
```

3. Get stack trace from any exception object:

```
try
{…}
catch (Throwable t)
{
        t.printStackTrace();
        throw t;
}
```

4. Use the reflection mechanism to enumerate all fields:

```
public String toString()
{
        java.util.Hashtable h = new java.util.Hashtable();
        Class cls = getClass();
        Field[] f = cls.getDeclaredFields();
        try
        {
            AccessibleObject.setAccessible(fields, true);
            for (int i = 0; i < f.length; i++)
                  h.put(f[i].getName(), f[i].get(this));
        }
        catch (SecurityException e) {}
        catch (IllegalAccessException e) {}

        if (cls.getSuperclass().getSuperclass() != null)
            h.put("super", super.toString());
        return cls.getName() + h;
}
```

5. Don’t even need to catch an exception to generate a stack trace, use the following statement anywhere in your code to get a stack trace:

```
new Throwable().printStackTrace();
```

6. You can put a main method in each public class. Inside it, you can put a unit test stub that lets you test the class in isolation. Make a few objects, call all methods and check that each of them does the right thing.

Assertions (1)

- It often happen that your code relies on the fact that some of the variables have certain values
  - Integer index are supposed to be within certain limits
  - Object references are supposed to be initialized
- It is good to occasionally check these assumptions

```
public void f(int[] a, int i)
{
 if ( !(a != null && i >= 0 && i < a.length))
        throw new IllegalArgumentError
              ( “Assertion failed);
}
```

Assertions (2)

- Use assertions to do occasionally check:

```
public void f(int[] a, int i)
{
  Assertion.check (a != null && i >= 0 &&
          i < a.length);
}
public class Assertion
{
  public static void check(boolean condition)
   {
         if (!condition) throw new
IllegalArgumentError
                ( “Assertion failed);
   }
}
```

How to Remove Assertions Code?

- Use a static final variable that is set to true during debugging and to false when the program is deployed:

```
public void f(int[] a, int i)
{
  if (debug)
        Assertion.check (a != null &&
              i >= 0 && i < a.length);
  …
}
```

- Better solution: put the test into a separate class and not ship the code for that class with the release version – anonymous class:

```
public void f(final int[] a, final int i)
{// debug is not necessarily  to be static final
    if (debug)
     new Assertion()
     {     {check (a != null && i >= 0 &&
           i < a.length);}
     }
}
```

- new Assertion() { … } create an object of an anonymous class that inherits from Assertion. This class has no method, just a constructor. The constructor is written as an initialization block.
- new Assertion(){{check (…);}}; When debug value is true, then the compiler loads the inner class and constructs an assertion object. The constructor calls the static check method of the Assertion class and tests the condition. If debug is false, the inner class is not even loaded – the inner class code need not to be shipped with the release version of program.

Debug with JDB Debugger

- JDK includes JDB, an extremely rudimentary command-line debugger
- Idea of using JDB: set one or more breakpoints, then run the program. The program will stop at the breakpoint, then you can inspect the values of the local variables.
- To use JDB
  - Compile your program with –g option

    javac –g myJava.java
  - Launch the JDB debugger:

    jdb myJava

Project

Some student information including student name, student number, gender, birthday and GPA is store in data.txt file. Each line holds a record, and the columns of record are separated by comma. Some extra blanks might exist between columns. The constraint of each field is the following:

Student name: a string quoted by double quote
up to 30 letters, blanks might exist.

Student number: an positive integer up to 7
digits.

Gender: a character either 'F', 'f', 'G' or 'g'
quoted by single quote

Birthday: a string with format mm/dd/yyyy

GPA: a floating number with up to two
fraction digits

The name of date file is passed as command line parameter. Write a Java program that read data from this file and do the following process:

1. Give out

(1) The percentage of female students.

(2) The number of students who are older than 30.

(3) The average GPA of all students.

2. List information of the students whose GPA is equal or greater than 8.5 to the screen neatly. Use upper case letters for gender.

Some data might not be correct in the file. For example, the string of student name might be two long, the birthday is in wrong format, the GPA has more than two fraction digits, etc. Handle this situation in your program using exception handling mechanism.