Java Tutorial - 2000

Part II: Java Language and Object-Oriented Concepts

Instructors: Geoffrey Fox ,
Nancy McCracken
Syracuse University
111 College Place
Syracuse
New York 13244-4100

Java Language Basics

Java Language Basics

- Java syntax has many similarities to C/ C++.
  - All variables must be declared
  - Syntax, comments, control structures are the same
- But there are some differences
  - No malloc or free - it has automatic garbage collection
  - No pointers - designers felt pointer arithmetic not robust or safe
  - Can declare variables almost anywhere as needed.
  - No struct, union, enum, typedef from C - it has classes and objects instead.
  - Java characters are based on 16--bit wide Unicode Worldwide Character Encoding rather than the usual 8--bit wide ASCII. This allows full support of all alphabets and hence all languages
  - Primitive types for integers and floats have machine independent semantics
  - Booleans in Java have value “true” or “false” (not 0, 1, . . .)

Java Language Syntax

- Three types of comments are supported:
  - / / ignore all till the end of this line
  - / * ignore all between starts */
  - / ** an insert into an automatically generated software documentation */
    - » for / ** */ one inserts HTML documentation with some simple macros such as @see (to designate see also) BEFORE the method or class being documented
- Java reserves the following keywords:
  - abstract boolean break byte case catch char class const continue default do double else extends final finally float for goto if implements import instanceof int interface long native new package private protected public return short throw throws transient try void volatile while
  - Note goto is not allowed in Java but its still reserved!
- null, true, and false are literals with special meaning

Java Language -- Program Structure

- Source code of a Java program consists of one or more compilation units, implemented as files with .java extension.
- Each compilation unit can contain:
  - a package statement
  - import statements
  - class declarations
  - interface declarations
- Java compiler (called javac) reads java source and produces a set of binary bytecode files with .class extensions, one for each class declared in the source file. For example, if Foo.java implements Foo and Fred classes, then "javac Foo.java" will generate Foo.class and Fred.class files.
- Suppose that Foo implements an applet and Fred is an auxiliary class used by Foo. If Netscape/ Internet Explorer encounters a tag <APPLET code="Foo.class">, it will download Foo.class and Fred.class files and it will start interpreting bytecodes in Foo.class.

Java Types

- Each Java variable or expression has a definite type, given by a declaration such as
  - int i;
  - double x, y, z;
  - Color c;
- There are three "types" of types!
  - There are Primitive or Simple types such as ints or booleans which are built-in.
  - New composite types (objects) can be constructed in terms of classes and interfaces. The type of an object is its class or interface
  - Arrays we will see are a sort of "almost" object!

Primitive Types

- There are 4 integer types: byte, short, int, long of size 8, 16, 32 and 64 bits, respectively.
- float is 32 bits, double is 64 bits. Floating point arithmetic and data formats are defined by IEEE754 standard.
- characters are given by 16bit Unicode charset and represented as short integers.
- One can use casts for conversion such as
  - long l;
  - l = (long) i;
  - / / which can be explicit as here and sometimes implied (see later)
- Note booleans are either TRUE or FALSE -- they are not 0, 1 ,-1 ...

Java Language -- Types: Array

- Arrays are "true" or "first class" objects in Java and no pointer arithmetic is supported.
- Like other objects, an array must be declared and created:
  - int states[]; / / declaration
  - alternative syntax: int[] vec;
  - and then:
  - states = new int[128]; / / creation
  - or concisely:
  - int states[] = new int[128];
- Arrays of arbitrary objects can be constructed,
  - e.g. Color manycolors[] = new Color[1024];
  - The only difference is that in the case of primitive types, the array elements are actually created. In the case of arbitrary objects, an array of object references is created; before you use array elements, you must call the constructor of that type for each element.

Java Language -- More on Arrays

- An array of length 128 is subscripted by integers from 0 to 127.
- Subscripts are range checked in runtime and so states[-1] and states[128] will generate exceptions.
- Array length can be extracted via the length instance variable, e.g.
  - int len = states.length will assign len = 128.
- Arrays can have dynamic sizing (a fixed size determined at runtime)
  - int sizeofarray = 67;
  - int states[] = new int[sizeofarray];
- Multidimensional arrays are arrays of arrays
  - » char icon[][] = new char[16][16];
  - » These arrays can be "ragged":
    - int graph[][] = new int[2][];
    - graph[0] = new int[4];
    - graph[1] = new int[7]; . . . graph[1][1] = 9;

Strings - an example of a class type

- Java provides many classes that represent data types, e.g. String. To declare a String variable and create a string:
  String s = new String (“This is the text.”);
- But in the case of strings, the special syntax is allowed:
  String s = “This is the text.”;
- Once created, individual characters of a string cannot be changed in place. The following example uses String methods to create a new string:
  - String test = "Chicken soup with rice";
  - int n = test.indexOf('w');
  - String newtest = test.substring(1,n-1) + "is n" + test.substring(n+5);
  - / *giving "Chicken soup is nice" */
- Comparing strings - use method equals instead of “==“
  test.equals (newtest)

Java Language -- Expressions

- Java's expressions are very similar to C and include the following forms. Both expressions and statements have values.
  - arithmetic:
    - » 2+3
    - » (2+3)*i
  - autoincriment and autodecriment
    - » i++ / * equivalent to i = i +1 */
  - boolean
    - » ((i > 0) && (j>0)) | | (state = -1)
  - bit operations
    - » i <<1 / * Left shift by 1 binary digit */
  - conditional expression
    - » (i>0) ? expression1 : expression2
  - strings have operators such as catenation
    - » "fred" + "jim" is "fredjim"
  - object property operators
    - » (a instanceof B) / * True iff object a is of class B */

Java Language -- Control Flow I

- if (some boolean expression) { .. }
  - else { ... } / / optional else
- Nested: if (some boolean expression) { .. }
  - else if(another boolean) {.. }
  - else { ... }
- while (any boolean) { / * Do Stuff */ }
- do { / * What to do */ } while (another boolean);
- for (expression1; booleanexpression ; expression2) {...}
  - naturally starts with expression1, applies expression2 at end of each loop, and continues as long as booleanexpression true
    - » for (int i=0; i<length(a); i++)
    - » a[i] = i * i;
    - » / * loop variable i is optionally declared to be local to loop as shown here. */

Java Language -- Control Flow II

- switch (expression) / * Just as in C */
  - {
  - case Constant1: / * Do following if expression=Constant1 */
    - » / * Bunch of Stuff */ break;
  - case Constant2: / * Do following if expression=Constant2 */
    - » / * Bunch of Stuff */ break;
  - default:
    - » / * Bunch of Stuff */ break;
  - }
- One can go to the next iteration of a loop by using
  - continue;
- or break out of the loop by using
  - break;

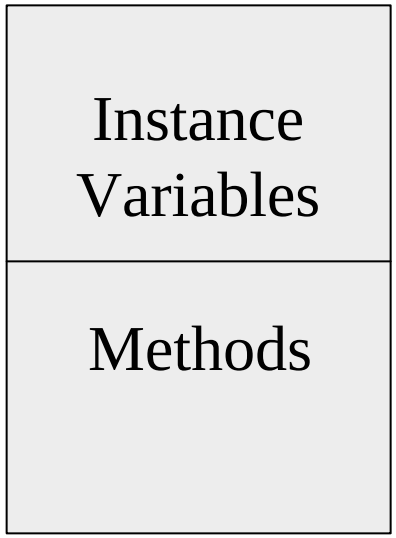
Method Definitions

- Subprograms in Java are called methods. The definition format is
  - Modifiers Returntype Methodname (Parameterlist)
    - » {
    - » declarations and statements
    - » }
- The parameter list contains the types and names of all the parameters.
- The declarations and statements are called the body of the method. Parameter names and variables declared in the body are local to it.
- Control returns from the methods either when the body is finished execution or a return statement is encountered. Return statements may also return a result.
- Parameters of primitive types are passed by value, of class types by reference.

The Java Object Model: Classes, Instances and Methods

The Java Object Model Overview

- Programs are composed of a set of modules called classes. Each class is a template specifying a set of behaviors on the data of the class.
- Each class has class variables (sometimes called instance vars) to hold the data and methods (called functions or procedures in other languages) to define the behaviors. Each object in a program is created as an instance of a class. Each class instance has its own copy of the class variables.
- Classes can be used for data encapsulation, hiding the details of the data representation from the user of the class (by marking variables as private).



Defining a Class

- The class definition consists of
  - a header line giving the class name, modifiers, possible subclass and interface structure
  - declarations (and possibly initializations) of class variables (aka instance variables)
  - declaration of a constructor method. This method has the same name as the class and does any initialization whenever an instance is created.
  - declarations of other methods.

API of a Class

- Each class has an API (Application Programming Interface) consisting of all the variables and methods that other programmers (i.e. in other classes) are allowed to use. These are designated by the "public" keyword.
- Example showing part of the Java Date class:
  - public class String
  - { / / Constructor methods to create instances of class
  - public Date ( );
  - public Date (long);
  - / / Accessor and Mutator methods to access and change data
  - public int getTime ( );
  - public void setTime (long);
  - / / Other public methods
  - public boolean after (Date);
  - public boolean equals (Date);
  - . . . }

Using a Class

- This declares object today to have type class
  - Date today
  - Date() is Constructor of Date class which constructs an instance of Date class and sets default value to be the current date
  - new Date()
  - Note that there are two constructor methods in this class, as in general Java allows overloading of methods (but not operators).
- An example application using a method of the Date class:

```
import java.util.Date;
class DateTest
{public static void main ( String[ ] args)
   {Date today = new Date();
    Date early = new Date(1000);
    if (today.after (early))
       System.out.println( "Today is not early!");
    }}
```

A Computational Class

- A class (such as a "main routine") may also be implemented to have just one computational instance.
- This application reads from standard input and counts number of characters which are then printed
  - class Count {
  - public static void main (String[ ] args)
  - throws java.io.IOException
  - { int count = 0;
  - while (System.in.read() != -1)
  - count++;
  - System.out.println("Input has " + count + " chars.");
  - }}

Header of Class Definition

- Class declaration in Java shares common aspects with C++ but there are also some syntactic and semantic differences.
- ClassModifiers class className [extends superClass] [implements interfaces] {<body of class>}
  - e.g. public class Test extends Applet implements Runnable { . . . }
  - defines an applet that can use threads which have methods defined by Runnable interface
- Only single inheritance is supported but aspects of multiple inheritance can be achieved in terms of the interface construct. Interface is similar to an abstract class with all methods being abstract and with all variables being static (independent of instance). Unlike classes, interfaces can be multiply-inherited.

Access Modifiers of Classes - I

- Possible ClassModifiers are:
  - abstract -- Contains abstract methods without implementation -- typically such abstract classes have several subclasses that define implementation of methods
  - public -- May be used by code outside the class package and (unix) file must be called ClassName.java where ClassName is unique public class in file
  - private -- this class can only be used within current file friendly(i.e. empty ClassModifier) -- class can be used only within current package
  - protected -- Only accessible to subclasses

Access Modifiers of Classes - II

- threadsafe: Instance or static variables will never change asynchronously and so can use compiler optimizations such as assigning to registers. Next modifier -- final -- is also valuable to compilers
- final -- Cannot have a subclass for classes
  - » cannot be overridden for methods
  - » final variables have a constant value e.g.
    - ◆ final int ageatdeath = 101;
- transient -- specifies that objects are not persistent
- Note most of these modifiers can be used either for a class or an object -- a particular instance of a class
  - » abstract only makes sense for a class and transient is perhaps more useful on an object basis

Access Modifiers of Methods

- MethodModifier ReturnType Name(argType1 arg1, ...)
- Returntypes are either simple types (int, byte etc.), arrays or class names
- Possible MethodModifiers are:
  - public -- This method is accessible by all methods inside and outside class
  - protected -- This method is only accessible by a subclass
  - private -- This method is only accessible to other methods in this class
  - friendly(i.e. empty) -- This method is accessible by methods in classes that are in same package
  - final -- a method that cannot be overriden
  - static -- This method is shared by ALL instances of this class and must be invoked with <Class>.method syntax.
  - synchronized -- This method locks object on entry and unlocks it on exit. If the object is already locked, the method waits until the lock is released before executing -- can be used on methods or statement blocks
  - native -- to declare methods implemented in a platform -- dependent language, e.g. C.

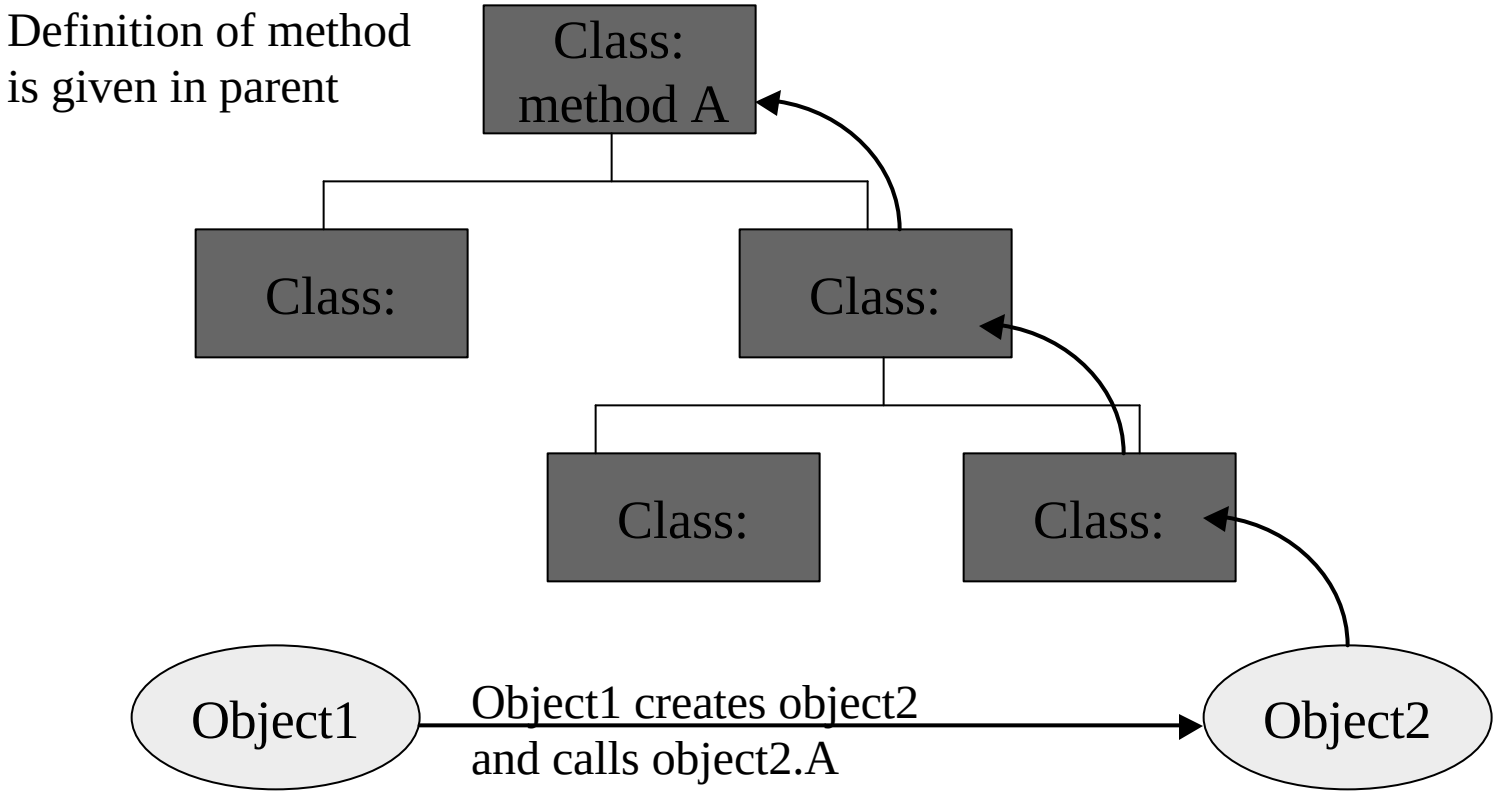
The Java Object Model: Inheritance and the Class Hierarchy

Relationships between Classes

- use
  - » A uses B: A calls a method (sends a message to) an object of class B or creates, receives, or returns an object of class B.
  - – containment
    - » A has a B: special case of use - an object of class A contains an object of B
- inheritance
  - » B is an A: specialization - B extends A (is a subclass of A) if B has all the variables and methods of A (and more).
  - – In the class definition of B, the child class, there is no need to repeat declarations of variables and methods of A, they are assumed to be there. The definition of B has the additional variables and methods of B.

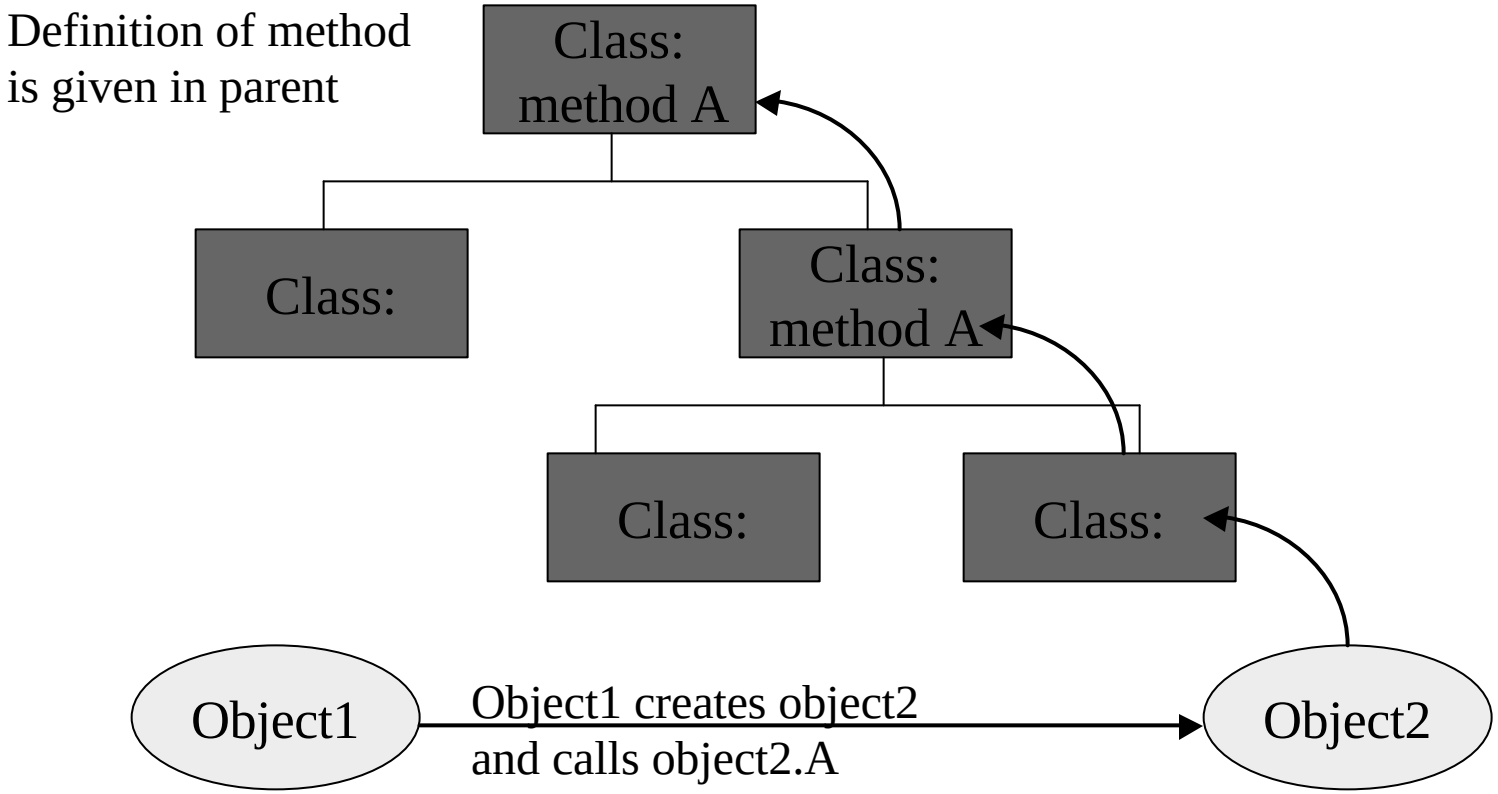
Use of Methods Defined in Parent

- If you call a method in a class that was defined in some parent, the compiler has a simple algorithm to find the definition: it searches up the parent/ child hierarchy.



Use of Methods Defined in Parent but overridden in child class

- This algorithm also works for the case when the method is overridden



Comments on Casting

- Casting (type conversion) is supported between types and class types. Syntax:
  - (classname)reference
- Two forms of casting are possible: widening and narrowing
- Widening, where the subclass is used as an instance of the superclass, is performed implicitly
- Narrowing, where the superclass is used as an instance of the subclass, must be performed explicitly
- Given Parent: Dot -> DrawableDot (Child):
  - Widening: An instance of DrawableDot is used as an instance of Dot
  - Narrowing: An instance of Dot is used as an instance of DrawableDot
- Casting between sibling classes is a compile-time error
- Note that otherwise conversions between types are given explicitly by methods within the class.

Array - A Pseudo Class!

- Not in any package
- One final instance variable: length
- For each primitive type (and all classes), there's an implicit Array subclass
- Cannot be extended (subclassed)
- Superclass is Object
- Inherits methods from Object
  - new int[5]).getClass().getSuperclass()
  - will return Java.lang.Object

Comments on Overloading and Overriding in Classes

- Overriding Methods (where child class provides method with same signature as method in parent)
  - To override a method, a subclass of the class that originally declared the method must declare a method with the same name, return type (or a subclass of that return type), and same parameter list.
  - When the method is invoked on an instance of the subclass, the new method is called rather than the original method.
  - The overridden method can be invoked using the super variable.
  - Super can be used to refer to instance variables in the superclass as well.
- Overloading (where a class can provide a set of methods all with the same name, but with different signatures): The signature is defined (as in Arnold-Gosling book) by
  - Lowest conversion cost of parameter list, based on type and number of parameters. Return type and declaration order not important.
  - Java will declare an error if method is invoked where there is not one with a unique signature

Abstract Methods and Classes Interfaces (classes without implementation)

Abstract Methods and Classes

- An abstract method has no body - it is provided in a class to define the signature of the method for program structuring purposes. It must be defined in some subclass of the class in which it is declared.
  - Constructors, static methods, private methods cannot be abstract
  - A method that overrides a superclass method cannot be abstract
- Classes that contain abstract methods and classes that inherit abstract methods without overriding them are considered abstract classes
  - It is compile-time error to instantiate an abstract class or attempt to call an abstract method directly.

Java Language -- Interfaces - Overview

- An interface specifies a collection of methods (behaviors) without implementing their bodies (akin to giving the API).
  - public interface Storable {
    - » public abstract void store(Stream s);
    - » public abstract void retrieve(Stream s);
    - » }
- Any other class which implements the interface is guaranteeing that it will have the set of behaviors, and will give concrete bodies to the methods of the interface.
- Interfaces solve some of the same problems as multiple inheritance, without as much overhead at runtime.
  - There is a small performance penalty because interfaces involve dynamic method binding.
- Interfaces can be implemented by classes on unrelated inheritance trees, making it unnecessary to add methods to common superclass.

Interface Example -- Implementing Storable

- A class may implement an interface, in which case it provides the body for the methods specified in the interface.
- interface storable has store and retrieve methods
  - public class Picture implements Storable {
    - » public void store(Stream s) {
    - » / / JPEG compress image before storing
    - » }
    - » public void retrieve(Stream s) {
    - » / / JPEG decompress image before retrieving
    - » }
    - » }
  - public class StudentRecord implements Storable {
    - » . . .
    - » }

Interfaces can be used as Classes in type specification

- Interfaces behave exactly as classes when used as a type.
- The normal type declaration syntax "interfaceName variableName" declares a variable or parameter to be an instance of some class that implements interfaceName.

```
» public class StudentBody {
»     Stream s;
»     Picture id_photo; / /  of interface storable
»     StudentRecord id_card; / /  of interface storable
»     . . .
»     public void register() {
»         save(id_photo);
»         save(id_card);
»       }
»     public void save(Storable o) {
»         o.store(s);
»       }
» }
```

Further Features of Interfaces

- Interfaces are either public or have the default friendly access (public for the package and private elsewhere)
- Methods in an interface are always abstract and have the same access as the interface. No other modifiers may be applied
- Variables in an interface are public, static, and final. They must be initialized.
- Interfaces can incorporate one or more other interfaces, using the extends keyword:
  - public interface DoesItAll extends Storable, Paintable {
    - » public abstract void doesSomethingElse();
    - » }
- A class can implement more than one interface:
  - » public class Picture implements Storable, Paintable {
  - » public void store(Stream s) {...}
  - » public void retrieve(Stream s) {...}
  - » public void refresh() {...}
  - » }

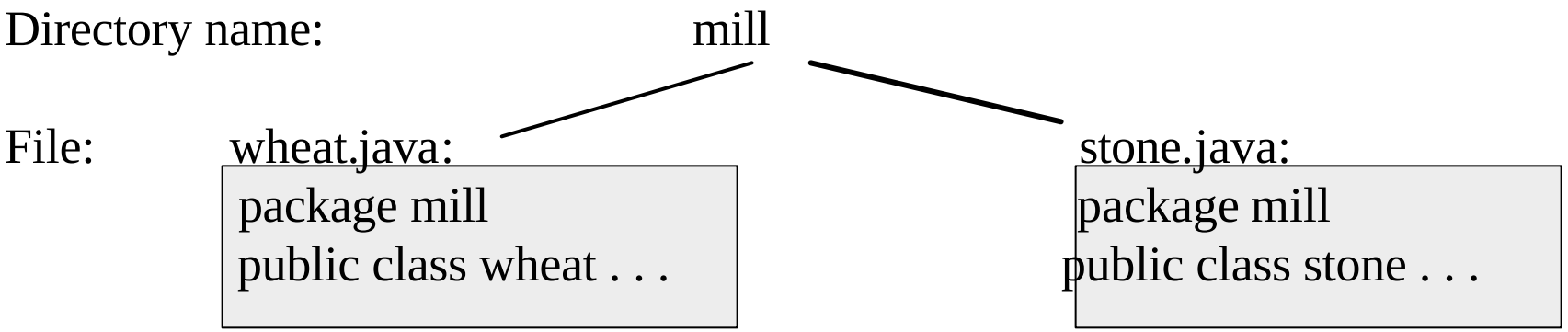
More on Interfaces -- Why use them

- Note that Interfaces often play a software engineering as opposed to required functional role
- Note that Interfaces are not significantly used in current Java release where perhaps there are 15 times as many class definitions as interface definitions
- But Interfaces play a crucial role in structuring programs that need to declare multiple sets of behaviors such as applets and threads.
- And Interfaces play a crucial role in the Remote Method Interface (RMI), where an Interface is the common specification between a Java applet or application and the set of methods that it can call remotely.

Packages in Java

Overview of Packages and Directory Structure

- One file can contain several related classes, but only one of them can be public. If the public class is called wheat.java, then the file must be called wheat.
- A set of classes in different files can be grouped together in a package. Each of the files must be in the same directory and contain the command
  - package mill;
- The name of the directory must the same as the package.



Using Java packages

- One conveniently uses files in a package by inserting
  - » import mill.*
- at the beginning of a file that needs classes from the mill package
  - Then classes in the mill package can be refered to by just using their Classname
  - ithout the import command, one must explicitly say mill.Classname
- Packages can be grouped hierarchically, with the corresponding directory tree. For example, the mill package could be a subpackage of agriculture. Then a class is referred to as agriculture.mill.Classname.
- Except for classes provided with the Java language, a class that is imported or used must either be in the current directory or be accessible to the compiler through the CLASSPATH environment variable.

Java 1.0 System Packages

- java.lang Contains essential Java classes and is by default imported into every Java file and so import java.lang.* is unnecessary. Thread, Math, Object and Type Wrappers are here
- java.io contains classes to do I/ O. This is not necessary (or even allowed!) for applets which can't do much I/ O in Netscape!
- java.util contains various utility classes that didn't make it to java.lang. Date is here as are hashtables
- java.net contains classes to do network applications. This will be important for any distributed applications
- java.applet has the classes needed to support applets
- java.awt has the classes to support windowing -- The Abstract Windows Toolkit
- java.awt.image has image processing classes
  - java.awt.peer is a secret set of classes with platform dependent details

Additional Java 1.1 System Packages

- java.awt.datatransfer Classes and interfaces to transfer data from a Java program to the system clipboard (enabling drag-and-drop)
- java.beans Contains classes to write reusable software components
- java.lang.reflect Enables a program to discover the accessible variables and methods of a class at run-time
- java.rmi Remote Method Invocation
- java.security Enables a Java program to encrypt data and control the access privileges provided
- java.sql Java Database Connectivity (JDBC) enables Java programs to interact with a database using the SQL language
- java.text Classes that provide internationalization capabilities for numbers, dates, characters and strings
- java.util.zip Combines java .class files and other files into one compressed file called a Java archive (JAR) file.

Additional Java 1.2 System Packages

- javax.accessibility - contracts between user interface components and assistive technology
- javax.swing - additional user interface components as well as providing standard “look and feel” for old ones
  - border, colorchooser, event, filechooser, plaf, table, text, tree, undo
- org.omg.CORBA - Provides the mapping of the Object Management Group CORBA APIs to the Java programming language, including the class ORB, which is implemented so that a programmer can use it as a fully-functional Object Request Broker (ORB).

More on the Java Language: Exceptions

Java Language -- Handling Runtime Errors Using Exceptions

- The language itself supports concept of an exception
- Java supports a hierarchical model of exceptions which allow and indeed require user to supply suitable handlers for any exception that can occur in a Java program
- Note exceptions that can occur in a method must either be caught (i.e. handled inside method) or thrown (i.e. returned to callee)
- Thrown exceptions are like returned arguments and are for instance part of interface to a method
- Exceptions are all (at some level in hierarchy) subclasses of Throwable class

The try statement for handling exceptions

- File file; / *defines file to be object of class File */
  - / *The body of the try statement is executed until an error occurs.
    It then skips to the body of the catch. */
  - try{
  - file = new File("filenameyouwant");
  - . . . . .
  - file.write("stuff put out");
  - } catch (IOException e) {
  - / / This catches ALL I/ O errors including read and write stuff
  - / *Handle Exception somehow */ }
  - return;
  - }
  - / * but the optional finally clause will be executed
  - whether or not code terminates normally */
  - finally
  - { file.close(); }

User Created Exceptions

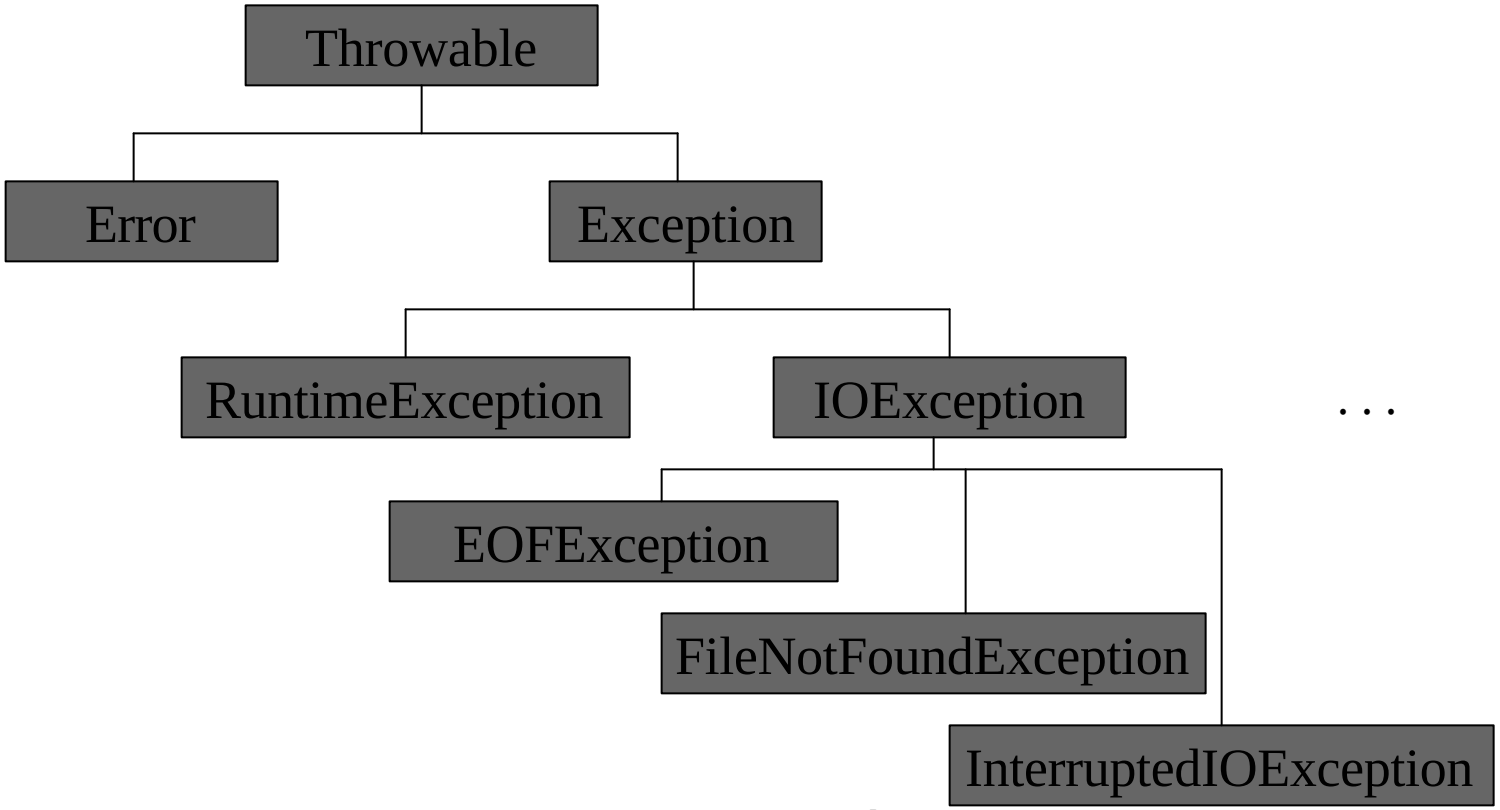
- The Exception class has data structures and methods that can give information about the exception and how it occurred. There are two constructors, one of which allows a message to be included in each instance.
- The user can either throw an exception of type Exception with a unique message, or create own subclass of Exception:
  - public static void MyMethod() throws MyException
  - { . . .
  - throw new MyException;
  - . . . }
  - class MyException extends Exception
  - {public MyException ()
  - {super ("This is my exception message."); }
  - }
- Methods which call "MyMethod" should use a try and catch block which catches an exception e of type MyException. Methods e.getMessage and e.printStackTrace can be used on Exceptions.

Basic Structure of Exception Handling in Nested Calls

```
– method1 {
–    try {
–         call method2;
–       } catch (Exception3 e) {
–         doErrorProcessing(e);
–       }
–  }
– method2 throws Exception3  {
–    call method3; / /  method2 just passes exception through
–  }
– method3 throws Exception3 {
–    call dividebyzeroorreadfileorsomething; / /  create exception
–  }
```

Examples of Exception Hierarchy

- As Examples of hierarchy:
- catch(FileNotFoundException e) { .. } would catch particular exception whereas
- catch(IOException e) { .. } would catch all IOexceptions



Classes of Exceptions

- There are two subclasses of Throwable
  - Error such as OutOfMemoryError which do NOT have to be caught as they are serious but unpredictable and could typically occur anywhere!
  - Exception which we have discussed
- Exception has a subclass RuntimeException that need NOT be caught
  - Typical RuntimeException subclasses are
    - » ArithmeticException, ClassCastException, IndexOutofBoundException
- Note that exceptions which are thrown but not caught appear as error message on stderr. For applets this is in the "Java console" of the browser.