CSE 413 - Au 02 - Programming Languages 11-Dec-2002
Final Java Review Sheet

Exam coverage

The final exam will cover material from the entire course. Much of what we have done since the midterm is apply what we learned before then, so all the material that was listed in the midterm Java review sheet is still appropriate for this final exam. Study the midterm Java review sheet in addition to this review sheet.

Files and Streams

Sources and sinks. Data read or written by a program is considered to come from a source and go to a sink. Sources and sinks can be files, memory, the console, network ports, etc. At the lowest level, every source and sink deals in bytes, but there are access methods provided called streams which can impose a higher level structure on the data while it is being read and written.

Streams. Getting the data from the source to the sink is the job of a stream object. Streams are subclassed and layered to provide whatever functionality is needed. There are two basic stream classes used for raw data: InputStream and OutputStream. These are abstract classes and provide only the most basic methods. The fundamental methods read() and write() are abstract and are implemented in the subclasses.

Stream subclasses. FileInputStream is an important subclass of InputStream that specifies a file as the data source. FileOutputStream is an important subclass of OutputStream that specifies a file as the data sink. These classes are concrete (not abstract) but they don't provide all the functionality that you might want at the application level.

Layered stream subclasses. Additional functions for a stream are provided by passing the stream object to the constructor of a "decorator" class. These classes take a stream as input to their constructor, and return a stream. The new stream has access to the methods provided in the decorator class. Examples of classes used in layering for input are BufferedInputStream, ObjectInputStream, and ZipInputStream. Examples of classes used in layering for output are BufferedOutputStream, ObjectOutputStream, and ZipOutputStream.

End of stream on read. An expected end-of-stream condition should be detected by checking for -1 returned as the number of bytes read. An unexpected end-of-stream condition will result in an EOFException which should be caught or thrown. An exception should only occur if something unusual happens, not just that you've real all the way through to the end of the file.

File class. The File class is used to specify and manipulate pathnames. There are several overloaded constructors for the File class, providing flexibility in how directories and filenames are specified. The File() constructors create new pathname objects, not new files. There are numerous methods available to operate on a File object, including methods to check for existence and access permissions (eg, exists(), canRead(), canWrite()), create and delete files (eg, createNewFile(), createTempFile(), delete()) and to get directory information (eg, getParent(), list(), listFiles()).

Exceptions

Exceptions are thrown to signify that an unexpected error has been detected. The relevant information about an exception is stored in an object of a subclass of Throwable. Exceptions are not meant to be used for simple, expected situations, since they are more costly than simple checks like "if (var==null) {... do something ...}".

Errors and Exceptions. There are two primary types of Throwable objects. The class Error is for problems that are serious and that cannot be fixed by the application program. Error exceptions include LinkageError, OutOfMemoryError and StackOverflowError. Programs should not attempt to catch Errors. The Throwable subclass Exception is for problems of a less serious nature, which the program either may be able to fix, or should have prevented in the first place.

Exceptions and RuntimeExceptions. One subclass of Exception is RuntimeException. This class of problem does not have to be explicitly declared or caught. The programmer's job in this case is to make sure that these problems don't occur in the first place. RuntimeExceptions include IndexOutOfBoundsException and NullPointerException. All other subclasses of Exception must be explicitly declared or caught and dealt with. These are checked exceptions. If you call a method that might throw an Exception, but you have not announced that you will throw it or you have not caught it, the code will fail to compile. Examples of Exceptions that are not RuntimeExceptions include FileNotFoundException and DataFormatException.

A method declares that it might throw an Exception in the method header. The keyword throws is used after the parameter list, followed by the name of the Exception class. For example, a method could be declared as follows:

```
public void addFileData(String s) throws IOException {...}
```

Try/catch. The code that is using a method that might throw an exception must be prepared for the event. The calling method can either state that it may throw an exception as discussed in the previous item, or it can catch the exception when it happens. In order to do this, it needs to surround the possible problem call with a try/catch block. The syntax is

```
try {... possible problem ...}
catch (<ClassOfException> e) { ... deal with problem ...}
finally { ... executed no matter what happens, including an Exception for which there is
no catch ...}
```

Multiple catch blocks are allowed, and are searched in order from top to bottom if an Exception occurs. Optionally, an additional block can be specified using finally, which will be executed after the try block, even if an Exception occurs for which there is no catch block.

Methods defined in the Throwable class. There are useful methods defined in Throwable which are inherited by all Exception classes, primarily related to printing the message associated with an Exception, and the stack trace. These methods can provide useful debugging information.

It is possible to extend the Exception class for your own specific error conditions. The subclass can just be a new name or it can define added variables, methods, and constructors as needed.

Packages

Import and package statements. The import statement is used to tell the compiler where to find classes that you have referred to by class name only. It does not include code in the compilation, it just tells the compiler how to find the class files it needs. The package statement tells the compiler that a class is part of a particular package, and thus the class files can be found according to a particular directory structure.

Static variables and methods

An application starts executing in the "main" method of the class that is specified to the java tool. The main method is always specified as "static void main(String[] args)" and the user command line options are passed to the program in the String array "args".

Static variables and methods are defined once when a class is loaded, and they are not repeated when an object is created. This can be useful for defining utility methods that are not associated with any one object, and also for creating a static variable that applies to all members of the class, rather than to each individual object of the class. Static methods and variables are accessed by preceding the member name by the name of the class, rather than the name of the object reference. Thus, to access a search method in the Arrays class, you could type Arrays.binarySearch(a,key), and to access the maximum Integer value possible you could type Integer.MAX_VALUE.

Reflection

The reflection API allows us to manipulate classes in a general sense without knowing at the time we write the code which classes will be involved. This capability is useful when building general purpose tools that can be extended with plug-in classes whose properties can be discovered and utilized at runtime, as well as in application builders that let us select classes from an existing and dynamic set to build a new application.

Every loaded class in an application is represented internally in the JVM by an instance of java.lang.Class, and we can access that using the getClass() method supplied by the class java.lang.Object and available for every object in a Java program.

Once we have a reference to a class object, we can use the Class method getName() to get the name of the class, and the method getSuperclass() to get a class object representing the superclass of this class. Other class information such as implemented interfaces and defined constructors, fields, and methods can also be obtained.

There are simple ways to create a new instance of an class, given the Class object for the desired class and using the default (zero-argument) constructor. Calls to other constructors that take arguments can also be constructed and called dynamically.