Welcome to: **Advanced Java**

XMaLpha Technologies, LLC.,

http://XMaLpha.com

Advanced JAVA (HANDS-ON)

Technical

This course picks up where Introduction to Java Programming leaves off. It covers Exceptions, Inner Classes, Collections, Streams, Packages, and JavaDoc (time permitting, Junit is introduced in this class.)

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Exceptions	Morning • Collections
Afternoon • InnerClasses	Afternoon • Streams

- And along the way:
  - Packages
  - JavaDocs
  - & various demos

XMaLphaTechnologies
Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Java Exceptions

- Exceptions vs Traditional Error Handling
- Types of Exceptions
- Calling Methods That Throw Exceptions
- Rethrowing Caught Exceptions
- Exception Methods
- Throwing Existing Exceptions
- Defining Your Own Exceptions
- Creating Hierarchies of Exceptions
- Overriding Methods that Throw Exceptions

Exceptions vs Traditional Error Handling

- Traditionally, functions return an error code if they fail
  - not always possible
  - does not follow OO paradigm
- Java uses Exceptions, which are objects
- Many methods in the Java library throw exceptions
  - must learn how to handle them
  - can throw them in your code
  - can choose to define your own

Probably the most important thing to learn about exceptions is how to call methods that throw exceptions. There are many methods in the Java library that throw exceptions, and if you want to use them, you need to understand what type of exception(s) they throw, and how to write code to deal with them.

Exceptions are objects in Java. You can look at the API to browse through the exception hierarchy.

Beyond the basics, the next step is to learn how to throw exceptions in your own code. The easiest way to do this is to throw exceptions that are already defined. Of course, you can also choose to define your own exception classes and throw those instead.

An exception is an object that encapsulates the exception. It often includes a message that goes along with the exception. There are also methods that can be invoked on the exception, which we'll see soon.

Types of Exceptions

- Root of exception hierarchy is Throwable
- Subclasses of interest
  - Error
  - Can't do anything about these
  - Exception
    - RuntimeException
    - Code should not cause these
    - Can catch them if desired, but not forced to do so
    - Other Exceptions
    - Must catch these - enforced by compiler

All exceptions descend from Throwable. There are several subclasses of this that are of interest.

The Error subclass is for internal errors, like running out of memory. There isn't much you can do about this, and you should not throw Error(s) yourself.

The Exception subclass also breaks into 2 parts, those that are RuntimeExceptions, and those that are not. RuntimeExceptions are considered to be in your control, meaning you should not write code that can throw them. For this reason, they are *unchecked* exceptions, meaning that the compiler will not force you to catch them. Since these are the easiest exceptions to force, we will be using them in examples. All other Exceptions are *checked* exceptions, meaning that the compiler will force you to handle them in your code.

NOTE: A method must declare all of the checked exceptions it may throw. Your code will not compile otherwise.

If an exception occurs and is not caught, it will terminate a non-graphical application and print a message to the console. For graphical applications and applets, it will print the message to the system console, and then return to the user interface.

Calling Methods that Throw Exceptions

- When calling a method that throws an exception, you have 2 options:
  - Catch and handle the exception
    - requires try-catch block
    - can also have finally block
  - Advertise that your method also throws the exception
    - requires throws clause
    - can specify that it throws
      - the same exception
      - parent of the exception

You have 2 options when you call a method that throws an exception. You can catch the exception in your method and handle it appropriately. (Actually, there's nothing to force you to handle it appropriately, but you should.) We'll see an example of this shortly. Or, you can choose to pass the exception on to whatever calls your method by throwing the exception as well. This requires a *throws* clause:

```
public void myMethod throws SomeException {
    // method definition here
}
```

You can specify that your method throws the exact same exception type as the method that you are calling. Or, you can specify that it throws some parent of that exception type. Think back to inheritance: If it throws IOException, then to say that it throws Exception is also correct, since IOException is-a Exception.

Simple try-catch Block Format

```
try {
    // call to method that may throw exception here
}
catch ( same-exception-or-parent-type-here id) {
    // handle exception here
}

Example:
try {
    String s = null;
    System.out.println("length = " + s.length());     //throws exception
}
catch (NullPointerException e) {          // could use Exception instead
    System.out.println(e.getMessage());
}
```


Page - 7

Keywords: try, catch, finally

In this example, we force an exception to be thrown by trying to use a reference that does not refer to an object. This will throw a NullPointerException. We can write the catch block to catch this exact exception, or to catch a parent of this exception, like Exception. Again, if it is a NullPointerException, then it is-a Exception.

The catch block is **only** executed if an exception is thrown. If execution proceeds without an exception, execution resumes after the catch block. If an exception occurs, the try block is terminated at that point, and the catch block is executed. Assuming you don't do something in your catch clause to leave the method, execution will continue after the catch clause.

Rethrowing Caught Exceptions

- If you catch an exception, but can't handle it completely
- Use a throw statement in the catch clause:

```
catch (SomeExceptionType e) {
   // do some cleanup
   throw e;          // re-throw exception
}
```


You may decide to do some cleanup in the catch block and then pass the exception on to the caller. The syntax for this is the same as throwing any other exception, except that you don't have to create an exception. It was passed into the catch clause.

Adding more clauses

```
try {
    // call to method that may throw exception here
}
catch ( same-exception-or-parent-type id1) {
    // handle exception here
}
catch (different-exception-or-child-type id2) {
    // handle this type of exception here
}
...
finally {
    // execute this whether or not exception occurred
}
```

It is possible to have more than one catch clause, if the code in the try clause can potentially throw more than one type of exception. However, the catch clauses must be in a specific order. You must catch the most specific exceptions first (back to that inheritance thing again). Otherwise, the first catch clause could catch anything (if it were set up to catch Exception, for example), and the other catch blocks would be unreachable. So, don't put a parent Exception before a child. BTW, the compiler will tell you if a catch block is unreachable.

The finally clause is optional, and often isn't necessary, but it can be useful instead of replicating code in both the normal execution, and the catch block. The finally clause is executed whether or not an exception occurs. If you catch an exception and don't do anything to exit the method, the finally clause is unnecessary, since execution will continue after the catch block anyway. It may be necessary if you return, exit, or rethrow an exception in the catch block.

NOTE: While we catch a RuntimeException here to illustrate the example, you don't often do this. Typically you can write your code to avoid these types of exceptions, although sometimes it's easier to catch them. See the ParseIntegers.java example for this.

Exception Methods

- getMessage()
  - returns a String containing the message associated with the exception
- toString()
  - returns a String which is the concatenation of the exception class name and the exception message
- printStackTrace()
  - prints a path of how you got to the code that threw the exception
  - especially useful when dealing with heavily nested code

You may have noticed that the catch blocks actually have to give an identifier for the exception that was thrown. These are methods that you can invoke on that exception object in the catch blocks. They are most useful for debugging.

Throwing Existing Exceptions

- Can choose to throw exceptions even if not calling methods that throw exceptions
- Method can throw more than one type of exception
- Options:
  - Determine which existing exception to throw
  - Define your own exceptions to throw
- Format:
  - throw new SomeExceptionType(“Optional Message”);

You can write your own methods to throw exceptions as well. If you encounter a problem that you cannot handle, it may be the best way to deal with it. This is especially true if you are writing a utility that takes data in, and is supposed to work on it. If the data is not as expected, you may want to throw an exception. This would be analogous to writing the Integer.parseInt() method.

It is also possible to throw multiple exceptions if your code warrants it. This would be preferable to throwing a single generic exception if there were several different things that could go wrong.

The easiest way to do this is to throw exceptions that are already defined in the Java library. Of course, if these don’t suit your needs, you can define your own as well.

NOTE: When a method throws an exception back to the caller, it does not continue execution, and hence, does not return any value, even if a return type is specified. The line that called the method does not get reactivated. Instead, the catch block is the next thing to be done.

The code to throw an exception is simple:

```
throw new SomeExceptionType(“Optional message here”);
```

Replace SomeExceptionType with the type of exception you want to throw. This could have been done in 2 steps, separating out the exception construction from the throw statement.

Defining Exceptions

- If existing exception classes are not adequate
- Defined like any other class
- Should extend Exception, or one of its subclasses
  - Compiler will enforce catching
- Typical to provide 2 constructors
  - no arguments
  - String argument for the message

When you are defining your own exceptions, you are really just specifying a new category of exceptions. Although you can add additional functionality to your new class, often we don't. Instead, we are just creating a new type of exception. In this case, the code is simple:

```
public class MyException extends Exception {
   public MyException()   {}
   public MyException(String message) {
      super(message);
   }
}
```

You can extend any exception class that you like. For example, if you are processing files and you encounter a bad file format, you may want to create an exception class for this, perhaps called FileFormatException. In this case, you would probably want to extend IOException since it is the parent of all exceptions dealing with input and output.

Exception Hierarchies

- It is possible to build an inheritance hierarchy of exceptions
- Same syntax as class hierarchies

```
public interface ChildException extends ParentException {
  … }
```

It is also possible to create your own hierarchies of exceptions, just like any other classes you may choose to define. The syntax is the same as defining any other inheritance hierarchy.

Overriding Methods that Throw Exceptions

- Can throw the exact same exceptions
  - type
  - number
- Can throw less exceptions
- Can throw children of parent's exceptions
- Cannot throw more exceptions than parent
- Cannot throw exceptions that parent doesn't throw
- Constructors are special, again

It is worth taking a look at how to override methods that throw exceptions. The restrictions on this slide are necessary to enforce normal processing in the situation where you have a parent reference to a child object.

For constructors, it is not possible to put a call to super() in a try block. Therefore, if the parent constructor throws an exception, the child constructor must also throw an exception. Exceptions are handy for constructors, since they cannot return values, and hence, can't return an error code.

Tips for Using Exceptions

- Exception handling is not supposed to replace a simple test
- Do not micromanage exceptions
- Do not squelch exceptions
- Propagating exceptions is not a sign of shame

Again, these hints are from the Core Java book by Horstmann and Cornell.

If you can take care of a potential problem before the exception could even occur, then do so. It is much more time-consuming to handle exceptions than to just execute a simple conditional.

If you wrap each line of code in a separate try-catch block, you will end up with a lot of code. Consider wrapping an entire task in a try block, even if it means having multiple catch blocks after it.

You see a lot of squelching going on in sample code, but it is not a good idea in practice. It is tempting to write an empty catch block just to get the code to compile, but you shouldn't do this in real code. Deal with the exception appropriately.

If the caller of your method is in a better position than your method to handle the exception, then propagate it up.

Exercise

- Write some code that causes a variety of runtime exceptions. Put statements after the exceptions are thrown to verify that these statements don't get executed. Use a finally clause, and see when it gets executed. Put an output statement after the finally clause, and see when it gets executed. Run your code several different ways, either by taking in arguments off the command line, or by changing your code and re-running it.

An alternative or additional exercise that is tied into the digital audio application is listed in the exercises file.

- Any questions?

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Welcome to: **Advanced Java**

XMaLpha Technologies, LLC.,

http://XMaLpha.com

Advanced JAVA (HANDS-ON)

Technical

This course picks up where Introduction to Java Programming leaves off. It covers Exceptions, Inner Classes, Collections, Streams, Packages, and JavaDoc (time permitting, Junit is introduced in this class.)

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Exceptions	Morning • Collections
Afternoon • InnerClasses	Afternoon • Streams

- And along the way:
  - Packages
  - JavaDocs
  - & various demos

XMaLphaTechnologies
Consulting & Education

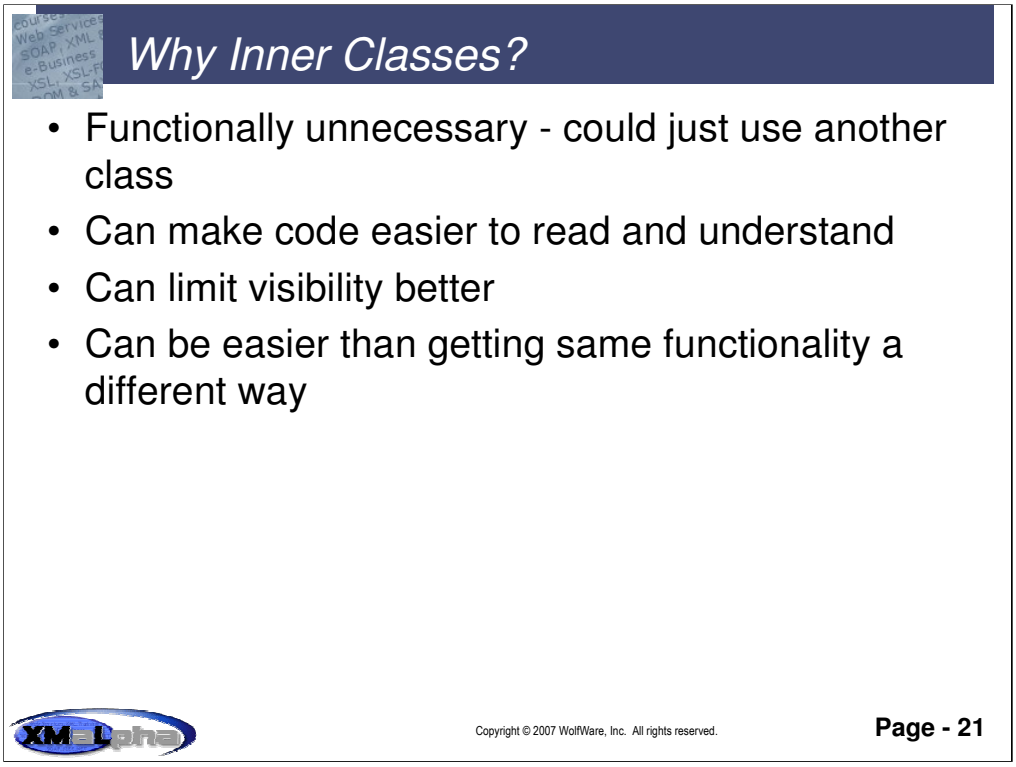
"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Inner Classes

- Why Use Inner Classes?
- Member Inner Classes
- Local Inner Classes
- Anonymous Inner Classes
- Static Inner Classes
- Example: Using an inner class as a true component object
- Example: Using an inner class when you need a different object



You can get the same functionality without using inner classes. However, they are sometimes easier to read, especially if the inner class is only needed temporarily.

Inner classes can be hidden from other classes in the same package, giving you more control over visibility. Since you aren't making the class available to anything else, it will also be easier to implement and maintain, since you don't have to worry about all the different ways it could otherwise be used.

Objects of some inner classes can access private data from the outer class object, making it easier than passing this data as arguments. This can be very handy. Since you don't have to use accessor and mutator methods, you might be able to not write them in the first place, especially if you don't want anything else using them.

NOTE: Inner classes are compiled into their own class files. So an outer class that also has an inner class will compile into 2 separate .class files. One for the outer class and one for the inner class. The name of the outer .class file will be as expected. The name of the inner .class file will be named:

OuterClassName$InnerClassName.class

In the case of anonymous inner classes, the part after the $ will be an integer, starting at one, and incrementing for each inner class in the outer class definition.

Types of Inner Classes

- There are 4 types of inner classes
  - Member
  - Local
  - Anonymous
  - Static
- Can make some examples work with any of them
- Each tends to fit a particular situation better

We will look at each one of these.

Member Inner Classes

- Can think of as another member of the outer class
  - nested the same as other members
  - functions most like a contained object
- Entire inner class definition sits at the top-level of the outer class
- Cannot have static members
- Syntactical difference from other inner classes
  - no static qualifier
  - not defined inside a method

Member inner classes can be private, package, protected, or private. Regular classes can only be package or public.

You need an object of the outer class type in order to create an object of the member inner class type. An object of a member inner class type gets an implicit reference to an object of the outer class type. If you are in an instance method of the outer class, you have *this* as the outer object. If you are in a class method of the outer class, you will need to create an outer object to create an inner object. The syntax for this is somewhat odd. To do this, use *new* as though it were a member of the outer class:

```
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner();
```

OR:

```
Outer.Inner inner = new Outer().new Inner();
```

Member inner classes are probably the easiest of the 4 types. They are particularly useful if you have sub-object that no one else needs. We'll see an example of this later. Since you can make these private, you can restrict access to them, and thus keep the implementation only as complex as your outer class needs.

Local Inner Classes

- A complete class definition inside a method
- No access qualifiers allowed
- Access is limited to the method in which it is defined
- Not a member of the outer class
- Cannot have static members
- Can access final variables from enclosing method
- Syntactical differences from other inner classes:
  - complete class definition
  - defined inside a method

You may decide that you want to limit visibility even further, and make an inner class local to a method. While this does limit access, the more useful benefit is to have the inner class definition very near the code that uses it. Of course, for this to be practical, you should make sure that no other methods in the outer class need to use the inner class.

The other reason you may decide to use a local class is if you need to access data from the method itself. Of course, you could always pass the data to a member class, but this may be easier. Local class methods can only access **final** data from the method. Since these are constants, the values can be copied into the local inner object, which is what is needed to make these work.

Anonymous Inner Classes

- Most cryptic syntax
- Handy for UI Event Handling
- Only useful if only need one object of this type
- Same restrictions as local inner classes
- Cannot have constructors
- Definition and Instantiation combined into single statement
- Unusual syntax distinguishes it from other inner classes

Probably the most difficult inner class to get used to is the anonymous inner class. It has the most cryptic syntax, and we'll see shortly that you must look closely to distinguish between creating an object normally, and creating/defining an anonymous inner class.

History Lesson: Inner classes were added as part of the Java 1.1 version. The main reason for this is that they make using the new event handling mechanism much easier. We'll see examples of this later, when we learn about events. For now, understand that inner classes are very popular for UI programming. In particular, anonymous inner classes for certain circumstances. Since much UI programming is now done via an IDE using drag and drop, you might not even realize that these are being used, or the IDE might not use them. This is more of an issue if you are writing your own UI code, which is a painful way to do it.

We'll look at the unusual syntax of these after we finish introducing all of the inner class types.

Static Inner Classes

- Looks like a member inner class
- Qualified with static
- Does not have a reference to the outer class object
- Can access static members of outer class
- Can create an instance of the inner class without having an instance of the outer class
- Syntactical differences from other inner classes
  - complete class definition, not inside a method
  - has static as a qualifier

Static inner classes look very similar to member inner classes. The only syntactical difference is the *static* qualifier. However, they are different from all of the other inner classes in that they don't have an implicit reference to an outer class object. Like any other static member, you don't need an object of the outer class to use it. So, you can create inner class objects without outer class objects.

Static inner classes can also have access qualifiers, just like member inner classes.

These are only useful if the inner class does not need to refer to anything in the outer class except static members.

Other names you may see used for this are "nested class" and "nested top-level" class.

To create objects of the inner class:

```
Outer.Inner inner = new Outer.Inner();
```

OR, if within outer class:

```
Inner inner = new Inner();
```

Inner Class as a Component

- Consider database transaction management
- May need to implement if
  - database doesn't support transactions
  - database access was setup to prohibit it
- Needs methods to:
  - start transaction
  - commit transaction
- Needs to keep track of operations that occur in between in case rollback is needed

Transaction management allows you to treat a series of database actions as a single transaction. If something goes wrong in the middle, the actions can be rolled back. Some databases don't support transaction management. Besides that, sometimes database connections are set up so that you can't use the transaction management that is provided with the database.

In the code example for this, each TransactionObject is a combination of the object that was communicated to the database, along with the action for how it was handled, such as ADDED, DELETED, UPDATED.

The code for this example is in TransactionManager.java. It uses an inner class to hold the TransactionObjects. The TransactionManager holds a Vector of theTransactionObjects. This is a case where TransactionObject is useless to any other class, so it is a good candidate for an inner class. Since it doesn't make any sense to have a TransactionObject without a TransactionManager, we can just make this a member inner class.

Inner Class as a Helper

- Typical examples concern UI programming
  - need object with specific capabilities to handle events
- We'll postpone this until the UI section
  - need to understand the rest of the UI code to have the inner class make sense
- Earlier Comparator examples could be done with an inner class

The typical examples given to show inner classes often deal with UI programming, since the two go hand-in-hand. Writing event handling code without inner classes would be even more painful than it already is. Since much of the UI code is now written by an IDE, you might not ever have to do this, but it's a good example anyway. We'll postpone this until we talk about event handling, since the example wouldn't make much sense now.

However, the earlier examples that we saw for Comparator had the SortCirclesComparator class implement Comparator. This works, but it doesn't make a whole lot of sense to advertise that SortCircles implements Comparator. You might have implemented a very specific comparison that could be troublesome if anything else used it. An inner class would work as well, and could be considered a better solution. SortCircles wouldn't have to implement Comparator and the comparison code could be closer to where the objects are actually used, making the overall file easier to navigate. This could be done using 2 separate classes in the same file as well.

Exercise

- Rework the SortCirclesComparator example to use an inner class instead. Since you need to use this inner class from main, a static member class would be the easiest choice.

- Any questions?

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Welcome to: **Advanced Java**

XMaLpha Technologies, LLC.,

http://XMaLpha.com

Advanced JAVA (HANDS-ON)

Technical

This course picks up where Introduction to Java Programming leaves off. It covers Exceptions, Inner Classes, Collections, Streams, Packages, and JavaDoc (time permitting, Junit is introduced in this class.)

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Exceptions	Morning • Collections
Afternoon • InnerClasses	Afternoon • Streams

- And along the way:
  - Packages
  - JavaDocs
  - & various demos

XMaLphaTechnologies
Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Collections

- Java 1.1 Collections
- Enumeration
- Java 1.2 Collections
- Iterator
- Collections Class

Collections are required as part of the Sun Java Certification Exam. Aside from certification, they are very useful tools that can save you a lot of programming time.

Java Collections

- Alternative to arrays for holding groups of objects
- Implementations for most common data structures
  - stack, queue, linked list, map, hashtable
- Contained items must be objects
- Some collections are ordered (Lists)
- Some collections cannot contain duplicates (Sets)
- Some collections require key-value pairs (Maps)
- Disadvantage - lose type information

Collections are now included on the Sun Java Programmer Certification Exam. There are many different collections, but the exam requires an understanding of 4 basic types:

a simple collection which does not enforce order and allows duplicates

a list which does enforce order and allows duplicates

a set which does not enforce order and does not allow duplicates

a map which supports searching on a key which must be unique

These features can be combined in various permutations as well. Collections also differ in how they are implemented. If you have taken a course on data structures, many of the Java collections will look familiar.

These classes were made possible because Java has a single-rooted hierarchy. Hence, by defining a linked list of Object, you can store any type of Object in the linked list. Since everything in Java is an Object, we can have a linked list of any object. We can also mix up the types of objects in the collection, though. We either need to keep it straight what's in the collection, or use Reflection to find out what the objects are.

Java Collections

- Arrays
  - good for random access
  - slow for insertion, deletion, and growing in size
- Linked List
  - good for insertion, deletion, and growing in size
  - slow for random access
- Tree
  - good for insertion, deletion, growing in size, searching
  - slow for random access
- Hashing
  - good for insertion, deletion, growing in size, excellent for searching
  - slow for indexed access, requires overhead

Specific implementations of collections most commonly fall into 4 basic categories: array, linked list, tree, and hash table. Each of these has advantages and disadvantages.

Arrays are good for fast random access, but poor for insertions and deletions. Arrays also cannot easily grow in size. An array can be searched somewhat efficiently, if the programmer knows the right algorithm.

Linked lists are good for insertions and deletions, but must do sequential access. They can easily grow in size. Lists do not need to be ordered, but typically work well for ordering. Linked lists are inefficient for searches.

A tree is good for insertions and deletions, and can also grow in size easily. A tree must be ordered. If constructed with some care, a tree can be searched very efficiently.

A hash table is an example of a map, as it requires a key-value pair. The unique key is used for access and searching. It is somewhat efficient for data access, and allows growth as well. Hash tables require some overhead for using the keys. Because of this, they are not considered worthwhile for small data sets.

Java Collections

- Parent Interface is Collection or Map
- List and Set Interfaces extend Collection
- Collection class name can tell you a lot:
  - HashSet - uses hashing, does not permit duplicates
  - TreeMap - ordered map
  - TreeSet - order set, no duplicates, uses tree for storage
- Choosing best implementation to use may require an understanding of data structures and computational complexity

The Collection API class hierarchy is somewhat complex. The root is actually an Interface, either Collection or Map. List and Set are interfaces that extend Collection. There are also several abstract classes that implement these interfaces, which are extended by classes that we actually use. The name of a collection can often tell you quite a bit about its implementation and features. However, choosing the most appropriate implementation for a given problem requires some understanding of common data structures and computational complexity.

The heavy use of interfaces in the Collections API makes it relatively easy to swap one Collection for another. To do this, you need to use the Interfaces as data types instead of the actual collection class.

Java 1.1 Collections

- Vector
  - probably most commonly used collection
  - basically an array that grows in size easily
  - updated for the 1.2 Collections
- Stack
  - LIFO structure, subclass of Vector
- Hashtable
  - typical map features, no null values, not in hierarchy
- BitSet
  - used for working with individual bits

There were 4 collections in Java 1.1. Actually, there really wasn't a Collection hierarchy then, so these are rather disjoint. However, Stack is a subclass of Vector.

Note that Hashtable is considered all one word, and the 't' is not capitalized.

Enumeration

- Java 1.1 Collections use an Enumeration to traverse the collection
- Can be used with Hashtable, Vector, and Stack
- Get the Enumeration by invoking elements():
  - Enumeration enum = myVector.elements();
- Methods:
  - hasMoreElements() - true if more elements
  - nextElement() - gets next element
- Superseded by Iterator

We won't dwell on the Enumeration interface because it is not recommended any longer, since it has been superseded by Iterator. Vector (and thus, Stack) have been retrofitted to use Iterator as well.

To use Enumeration, invoke the elements() method on the collection. This method returns an object of type Enumeration. From that object, you can traverse the collection. Suppose I have a Vector of Circles:

```
Enumeration enum = circles.elements();
 while (enum.hasMoreElements())
   Circle c = (Circle) enum.nextElement();        // cast is necessary
```

Since collections work on Objects, we have to downcast to assign the element to a variable of its true type.

Java 1.2 Collections

- Other collections added in 1.2
- Wide variety of implementations to choose from
- Lists
  - LinkedList, ArrayList, Vector
- Sets
  - HashSet, TreeSet
- Maps
  - HashMap, TreeMap

Java 1.2 added to the collection classes substantially. In fact, the hierarchy of Interfaces and abstract classes was added with 1.2.

Vector was retrofitted to implement the List interface in 1.2. Other concrete classes that implement List are LinkedList and ArrayList. The main difference between these is the computational complexity of various operations.

Iterator

- Replaces Enumeration Interface for Collections
- Defined in List and Set Interfaces
- Get Iterator by invoking iterator():
  - Iterator iterate = myLinkedList.iterator();
- Methods:
  - hasNext() - true if there are more elements
  - next() - returns the next element
  - remove - removes the last item retrieved
- Vector and Stack also implement Iterator

The Iterator interface replaces the Enumeration interface. It is defined in the List and Set interfaces, so anything that implements those interfaces will have Iterator capabilities.

The shorter method names are also considered an improvement. There is also a remove() method that will remove the last item that was retrieved from the collection. If the collection has been modified in any way after the Iterator was gotten, these methods will not work properly.

Since Vector was retrofitted to implement List, Vector and Stack also have Iterator capabilities.

An example of using an Iterator to traverse a Collection is in Iterate.java.

The Collections Class

- Utility class for Collections
- Similar to Arrays class
- Has methods for:
  - sorting (sort())
  - searching (binarySearch())
  - max, min (max() and min())
- Can specify means of Comparison for sorting and searching with a Comparator

Recall the Comparable interface from the Interfaces section. The utility class, Arrays, uses it to sort an array of objects. Well Arrays also makes use of a similar interface, called Comparator. You can use either for dealing with arrays, as long as you implement the corresponding method. Comparable requires a method called compareTo(), and Comparator requires a method called compare(). Their behavior is the same.

Similar to Arrays, there is a Collections class that has utilities for working on Collections. Collections also uses both the Comparable and the Comparator interfaces.

Comparator and Comparable

- Both are interfaces
- Arrays and Collections utility classes can make use of either
  - sort with just one argument uses Comparable
  - sort with an extra Comparator argument uses Comparator
- Comparable requires a functional compareTo() method
- Comparator requires a functional compare() method

For both Arrays and Collections, if you call the sort method that only takes an array or collection, it expects the objects in the array or collection to implement Comparable, and uses the compareTo method to sort. Both classes have another option, where you can specify a Comparator to do the sorting instead.

In many cases, it doesn't matter which way you go, but if you run into a situation where the objects implement Comparable, and you want a different comparison, you would use the method that takes a Comparator and send it your own comparison for the immediate task at hand.

We'll look at some examples using Collections in class. In particular, we'll compare 4 cases:

Arrays, using Comparable - We saw this in the SortCircles example in the Interfaces section.

Arrays, using Comparator - Implemented in SortCirclesComparator.java.

Collections using Comparable and Comparator - Implemented in Iterate.java.

- Any questions?

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

Welcome to: **Advanced Java**

XMaLpha Technologies, LLC.,

http://XMaLpha.com

Advanced JAVA (HANDS-ON)

Technical

This course picks up where Introduction to Java Programming leaves off. It covers Exceptions, Inner Classes, Collections, Streams, Packages, and JavaDoc (time permitting, Junit is introduced in this class.)

Instructor: Joyce Deeb,

Senior Consultant, XMaLpha Technologies, LLC

Joyce Deeb is a senior consultant with XMaLpha Technologies (http://XMaLpha.com), has considerable experience in software engineering, high-level application design and development, advanced computing techniques, strategic technology planning, curriculum development, and teaching. Her level of technical depth in application programming, Java & XML development, web-based applications, data warehousing, knowledge-based systems, and parallel processing is impressive.

Course Syllabus

Day # 1	Day #2
Morning • Exceptions	Morning • Collections
Afternoon • InnerClasses	Afternoon • Streams

- And along the way:
  - Packages
  - JavaDocs
  - & various demos

XMaLphaTechnologies
Consulting & Education

"Got Meta-Data?"®

XMaLpha Technologies is a premier provider of full-scale, industrial-strength, XML solutions. As experts in analysis, design, and implementation, the talented consultants at XMaLpha understand data integration using XML. Learn how XML can provide a totally extensible, easy-to-learn, and richly featured universal format for structuring data and documents that can be exchanged efficiently over the Web using .NET, Java, and other interoperable technologies. Whether your needs call for business-to-business solutions, sophisticated Web Services, Content Management Systems, end-to-end integration with legacy data, structured dynamic content generation, or education and knowledge transfer on the latest XML, Java or .NET technologies, XMaLpha can help.

http://XMaLpha.com

Streams

- Stream Basics
- Binary Output Streams
- Character Output Streams
- Binary Input Streams
- Character Input Streams
- The RandomAccessFile Class
- Serializing Objects
- The File Class
- The FileNameFilter Interface

Streams

- A general purpose input/output class
- Provide a common way of doing I/O, regardless of source/destination
  - network connection, monitor, keyboard, thread, file
- Follow the Decorator Pattern
- To get such flexibility, dealing with Streams can become complex
- There are over 60 stream classes in the Java Library
  - must import from java.io

Streams are the way Java handles input and output. The idea is that input and output will work the same, regardless of the source or destination.

The Stream classes are set up according to the Decorator pattern. The idea is to add functionality by attaching it to a barebones class. You can add as much functionality as you need. In Streams, you will start out with a barebones Stream class and add on functionality by using that Stream to construct a more functional Stream. This is called filtering.

There are a lot of different Stream classes, making it a pretty tough feature to wade through. However, we can take some cues from the names of the Stream classes:

Piped => Threads

Buffered => Efficiency

File => dealing with a physical file

Data => methods for dealing with primitives

Object => methods for dealing with serializing objects

Zip => dealing with compressed zip format

We'll look at a few basic streams and see examples of how to read and write binary data, character data, and objects.

Using Streams

- Choose the Stream to correspond to the basic task you are trying to do
- Determine what features you want to add to that Stream
- Create the Stream
  - Constructors typically throw IOException
- Use the Stream
- Close the Stream

We'll look at several examples of choosing streams for particular applications, and how to add functionality to them by filtering them.

For any Stream, closing is the same:

```
streamVar.close()
```

We should be sure to close Streams as soon as we are done using them.

Stream Characteristics

- 2 Types of Streams
  - Binary or Byte Streams
  - Character or Text Streams
- 2 Types of Operations (mainly)
  - Reading (Input)
  - Writing (Output)
- First step in choosing which Stream to use is to pick one of these 4 possibilities

One way to break down the group of Stream classes is to split it into streams that deal with binary data, and those that deal with character data. For the most part, streams either do input or output as well. So by determining which one of these 4 combinations you want to do, you have a starting point for choosing your Stream class.

4 Base Abstract Classes

- InputStream - base class for doing binary input
- OutputStream - base class for doing binary output
- Reader - base class for doing text input
- Writer - base class for doing text output
- Most other Stream classes descend from these
- Related Classes that are not in the Stream family
  - RandomAccessFile
  - File

These are abstract base classes, which contain low-level methods for reading and writing. To make use of these capabilities, use descending classes that implement higher-level methods that are easier to use. The InputStream and OutputStream classes have methods for reading and writing bytes. The Reader and Writer classes have methods for processing Unicode characters.

There are a couple of closely related classes as well. The RandomAccessFile class acts like a Stream, but it not part of the Stream hierarchy. It has capabilities for non-sequential file access.

Another useful related class is File. The File class provides representation for physical files and directories.

Binary Output Streams

- Start with a type of OutputStream
  - FileOutputStream if sending output to a file
    - Constructor requires a String or File object
- Add functionality
  - BufferedOutputStream for efficiency
    - typically want this
  - DataOutputStream for methods to output primitives
- The outermost Stream should be the one that has the methods you want to use

FileOutputStream has the capability to write binary data to a physical file. A common setup for writing binary data to a file starts with a FileOutputStream and then adds buffering for efficiency. Finally, the useful methods are in DataOutputStream, which has methods to output primitives.

The FileOutputStream constructor takes a String or File as an argument. There is also one to specify whether or not to append. By default, it does not append.

The BufferedOutputStream constructor can take a FileOutputStream as an argument. This is how we "decorate" a stream with additional functionality. These streams use a buffer so that each operation does not require accessing the drive. Data is written when the buffer is full, or when the stream is closed.

The DataOutputStream constructor can take a BufferedOutputStream as an argument. It provides methods for writing primitives in a portable binary format.

Binary Output Stream

- Example:

```
DataOutputStream dos = new DataOutputStream(
                           new BufferedOutputStream(
                            new
FileOutputStream("store.bin")));
```

  - The dos reference will allow you to use methods available to DataOutputStream:

    writeDouble(), writeFloat(), writeUTF(), writeInt(), …

    dos.writeDouble(myDoubleValue);

- Argument to constructor can contain a relative or absolute path, otherwise it's assumed to be in the current directory

Here is a common setup for writing binary data to a file. Although it is somewhat cumbersome to set up the streams, once this is done, the Stream variable is easy to use.

Take notice of the writeUTF() method. This allows us to write out a String that can then be read back in with a single readUTF statement. This is much handier than reading strings character by character.

See BinaryOutput.java for an example of this.

NOTE: The directory separator character is not consistent across platforms. If this is a concern, do not use a String literal that has a path in it. You have 2 options. You can create file objects successively for each directory and it's immediate subdirectory. You can also use a platform independent file separator to construct your path String. This is stored as an attribute in the File class:

File.separator

You may also want to know what the user's current working directory is:

System.getProperty("user.dir")

and the platform dependent line separator is in:

System.getProperty("line.separator)

BTW, println() automatically uses this property.

Character Output Streams

- Automatically convert Unicode characters to the local character scheme
  - for US, this is ASCII
- Start with a type of Writer
  - FileWriter if sending output to a file
    - Constructor requires a String or File object
- Add functionality
  - BufferedWriter for efficiency
    - typically want this
  - PrintWriter for methods to output text
    - println() and print()
    - Constructor requires an OutputStream or a Writer

The stream classes that write character data have built in functionality to convert the Java Unicode characters to the local character encoding scheme. These schemes correspond to local languages, and each JVM has a default encoding scheme. The stream classes write data based on that local scheme.

Unicode can represent most characters, but not the Chinese alphabet. Java also has the means to write UTF characters. UTF is an encoding scheme that will use 1, 2, or 3 bytes to write out a single character. For ASCII characters, it only uses a single byte, so it's very space efficient. For some foreign alphabets it requires 2 bytes, and for others it requires 3.

FileWriter has the capability to write character data to a physical file. A common setup for writing character data to a file starts with a FileWriter and then adds buffering for efficiency. Finally, the useful methods are in PrintWriter, such as println().

The FileWriter constructor takes a String or File as an argument. There is also one to specify whether or not to append. By default, it does not append.

The BufferedWriter constructor can take a FileWriter as an argument. It is analogous to the BufferedOutputStream.

ThePrintWriter constructor can take a BufferedWriter as an argument. It provides methods for writing text.

Character Output Streams

- Example:

```
PrintWriter pw = new PrintWriter(
                    new BufferedWriter(
                     new FileWriter("log.txt")));
```

- The pw reference will allow you to use methods available to PrintWriter:
  - print(), println(), …
- String argument to constructor can contain a relative or absolute path, otherwise it's assumed to be in the current directory

Here is a common setup for writing text to a file.

Note: PrintWriter replaces the PrintStream class. PrintStream was intended to do the same thing as PrintWriter, but there are some limitations in its implementation. In particular, it assume ASCII as the translation destination format. You should use PrintWriter instead of PrintStream. However, there are some leftover implementations of it. Most noticeably, System.out is a PrintStream object.

See the CharacterOutput.java file for an example of this.

Binary Input Streams

- Start with a type of InputStream
  - FileInputStream if getting input from a file
    - Constructor requires a String or File object
- Add functionality
  - BufferedInputStream for efficiency
    - typically want this
  - DataInputStream for methods to input primitives

This setup corresponds very closely to the Binary Output Stream setup.

Binary Input Streams

- Example:

```
DataInputStream dis = new DataInputStream(
                          new BufferedInputStream(
                          new FileInputStream("store.bin")));
```

- The dis reference will allow you to use methods available to DataInputStream:
  - readDouble(), readFloat(), readUTF(), readInt(), …
  - double myDoubleValue = dis.readDouble();
- String argument to constructor can contain a relative or absolute path, otherwise it's assumed to be in the current directory

Here is a common setup for reading binary data from a file. The read methods take a variable of the corresponding type as an argument and put the read data into that variable.

See BinaryInput.java for an example of this.

Character Input Streams

- Automatically convert the local character scheme to Unicode characters
- Start with a type of Reader
  - FileReader if getting input from a file
    - Constructor requires a String or File object
- Add functionality
  - BufferedReader for efficiency
    - typically want this
- BufferedReader actually gives us a useful method:
  - String readLine()

FileReader has the capability to read character data from a physical file. A common setup for reading character data from a file starts with a FileReader and then adds buffering for efficiency. Diverging from the pattern we've seen so far, BufferedReader actually gives us a method that we probably want to use. The readLine() method will read a line of text from input and return it as a String.

The FileReader constructor takes a String or File as an argument.

The BufferedReader constructor can take a FileReader as an argument. It is analogous to the BufferedInputStream.

Character Input Streams

- Example:
- BufferedReader br = new BufferedReader(
- new FileReader("log.txt"));
- The br reference will allow you to use methods available to BufferedReader:
  - String getLine()
- String argument to constructor can contain a relative or absolute path, otherwise it's assumed to be in the current directory

Here is a common setup for reading text from a file. The getLine() method will return you a line from the file as a String.

Reading Text from Standard Input

- Use an InputStreamReader as the base stream
- Pass System.in to its constructor
- Filter it with BufferedReader to get the readLine() method
- Example

```
BufferedReader stdin = new BufferedReader(
                       new InputStreamReader(System.in));
System.out.println("Enter a line of text");
String memo = br.readLine();
System.out.println(memo);
```

Reading data from the keyboard is an example of reading character data. We don't use a FileReader because we're not dealing with files. Instead we use a class called InputStreamReader and pass it System.in. This stream can then be passed to the BufferedReader constructor to get buffering and the readLine() method.

See the StandardInput.java file for an example of this.

A couple of utility classes that are useful with Character Input Streams are StreamTokenizer and StringTokenizer. These classes have methods for helping to parse a series of characters into tokens.

RandomAccessFile

- Can read or write to the same file
- Read and write operations do not need to be sequential.
- RandomAccessFile is not part of the Stream hierarchy
- Has the same API as DataInputStream and DataOutputStream
  - methods for reading and writing primitives

RandomAccessFile is not part of the extensive Stream class hierarchy. However, it has similar functionality to the Stream classes. Additionally, it allows operations to jump around in the file. We'll see that you can ask for the current location, move on, and then return to that location later.

You can use this class for use with physical files, but it does not apply to network connections, since these must be sequential access. For that, you need Streams.

RandomAccessFile

- Constructors take either a String or a File and an additional String indicating read-only (“r”) or read-write (“rw”)
- Methods:

  long getFilePointer()

  long length()

  void seek(long bytesIntoFile)

The constructors for RandomAccessFile take either a String or a File object, and a second String that indicates whether you want to open the file for reading only or for reading and writing.

The getFilePointer() method will return the current location in the file as a long which is the number of bytes into the file.

The length() method will return the length of the file in bytes.

The seek() method will move the specified number of bytes into the file, always starting from the beginning. If you call seek(), any subsequent read/write operations will occur at the new location in the file.

See RandomAccess.java for an example of this.

Serializing Objects

- The process of sending/receiving objects across a Stream
- Used to be a tedious task
- Java has 2 classes to support serialization:
  - ObjectInputStream
  - ObjectOutputStream
- These classes have methods for I/O of objects:
  - Object readObject()
  - void writeObject(Object obj)

The task of writing and reading composite objects has traditionally been a tedious task that required considerable design work and attention to detail. This troublesome task is provided for us in the Object Stream classes. These classes also have methods for reading/writing all of the primitive types, but their real strength is in reading and writing composite objects. These classes extend from InputStream and OutputStream.

The readObject() method reads the next object from the InputStream and returns it as type Object. Either you need to know what the object type is, or you need to use Reflection to get the type in order to downcast it for use.

The writeObject() method writes the given argument to the OutputStream.

Serializing Objects

- Requirements for using serialization methods:
  - Class must implement the Serializable interface
  - Contained objects must either be Serializable or declared as transient (a qualifier)
- Recall that Serializable is a tag interface
- Attributes qualified as transient will be ignored by the serialization process

In order to use the Serialization methods, the object's class must implement Serializable. All contained objects must also be serializable in order for the methods to work with them. If they are not serializable, they must be declared as *transient* so that the methods will ignore them.

Object Output Stream

- Example:

```
ObjectOutputStream oos = new ObjectOutputStream(
                         new FileOutputStream("obj.bin")));
```

- The oos reference will allow you to use methods available to ObjectOutputStream:

```
oos.writeObject(myObject);
```

- Object can be read back in using an ObjectInputStream:

```
ObjectInputStream ois = new ObjectInputStream(
                        new FileInputStream("obj.bin")));
Object myObject = ois.readObject();
```


The object serialization methods make a complex task quite simple, by doing all of the recursive serialization of composite objects and all of the bookkeeping for efficiency behind the scenes.

See ObjectOutput.java for an example of this.

The File Class

- Useful for working with Streams
- Represents a file or directory
  - methods will tell you which type it is
- Constructors take either the file name as a single String, or divided up into path and subpath. Can also take a File object and a subpath String
  - File(String filename)
  - File(String path, String subpath-or-file)
  - File(File path, String subpath-or-file)
- Useful methods for working with file systems

A class closely associated with Streams is the File class. The Stream classes are concerned with the contents of a file, while as the File class is concerned with the file's status in the file system. A File object can represent either a file or a directory. The class has methods that you can use to tell which one it is. The class has a nice selection of methods for working with file systems, such as finding out if a file exists, if it is writeable, renaming a file, and making directories.

Be sure to check out the Java API for File to see a complete list of the methods.

The File Class

- Constructor Examples:

```
File f1 = new File("myFile");
File f2 = new File("subdir\MyFile");
File f3 = new File("subdir", "MyFile");
File f4 = new File("subdir");
File f5 = new File(f4, "MyFile");
```

- Method Examples:

```
if (f4.exists() && f4.isDirectory() && f4.canRead())
   // returns an array file names in f4
   String [] contents = f4.list();
```

It's handy that we can use File objects to create Stream objects for a couple of reasons. We often need to validate a particular file before actually using it. The File methods allow us to verify the existence of the file/directory and check other information about it before using it to construct a Stream. Creating a file object does not create a physical file on the file system, although there is a method in File for doing so. However, we can also pass the file to a Stream constructor to do the same thing.

The list() method gives us an array of Strings which are the file names contained in the directory invoking object. It is an overloaded method, where we can also specify a means to filter this list.

The FileNameFilter Interface

- Works well with File objects that represent directories
- Filters the contents returned by list() so that you only get specific contents, like all of the .java files
- One method which tells which files to include:
  - boolean accept(File dir, String filename)
- You provide the implementation for accept() to filter whatever you want
  - return true to include the file

Think of the window that pops up when you go to open a file. It shows the contents of a directory, but only those files that match what's in the box at the bottom, like only show .txt files. FilenameFilters help us do the same thing. This interface requires us to write a method that indicates true if we want a particular item included in the directory contents listing. The criteria for including a file is completely dependent on how we write the accept() method. We pass an object of type FilenameFilter to the list() method in order to filter the contents.

The FilenameFilter Interface

- Example:

```
File f = new File("myDirectory");
if (f.exists() && f.isDirectory()) {
   FilenameFilter fnf = new FilenameFilter() {
      public boolean accept(File dir, String fn) {
            return (fn.endsWith(".java")) ? true : false;
         }        // end accept method
   }       ;     // end anonymous class definition
   String [] contents = f.list(fnf);
}                // end if statement
```

Recall that there are several ways to get the behavior needed by an interface. We can have our class implement the interface. Or we can have a helper class that implements the interface. This helper class can either be a separate class or an inner class. We can use any kind of inner class that suits our purpose. In this case, an anonymous inner class is handy, because it puts the filtering criteria code very near the code that uses it. As long as this is the only place where this interface behavior is needed, this will work fine. Suppose this method needs to filter the directory contents several different ways. You couldn't handle that by having the class implement FilenameFilter, because then you could only write a single accept() method. You could do it with helper classes, but you would need one for each type of filtering. The easiest way is to repeat this code fragment for each type of filtering.

Recall that an anonymous inner class combines instantiation (*new*) with the class definition. Since the class has no name, the next thing after *new* either indicates the class it extends or the interface it implements. In this case, it implements FilenameFilter. The next thing is the body of the anonymous class, which only contains one method, accept(). The accept method uses the conditional operator to tell whether to return true or false.

This code is typically somewhat obscure for someone just becoming familiar with Java. However, it is a common way to get interface behavior that you only need to use once.

See FileOps.java for this example.

Exercises

- Make Track and Program Serializable. Their descendents will inherit this behavior.
- Write a driver class to test out this functionality by creating a program, serializing it, and reading it back in.

- Any questions?

A place for your notes…

1 "The Good, The Bad, and The Ugly"™ is a trademark of MGM Home Entertainment who current own the intellectual property rights to the 1966 movie by the same name, starring Clint Eastwood.

</Presentation>

- Please feel free to E-Mail the courseware author at:

Thank you

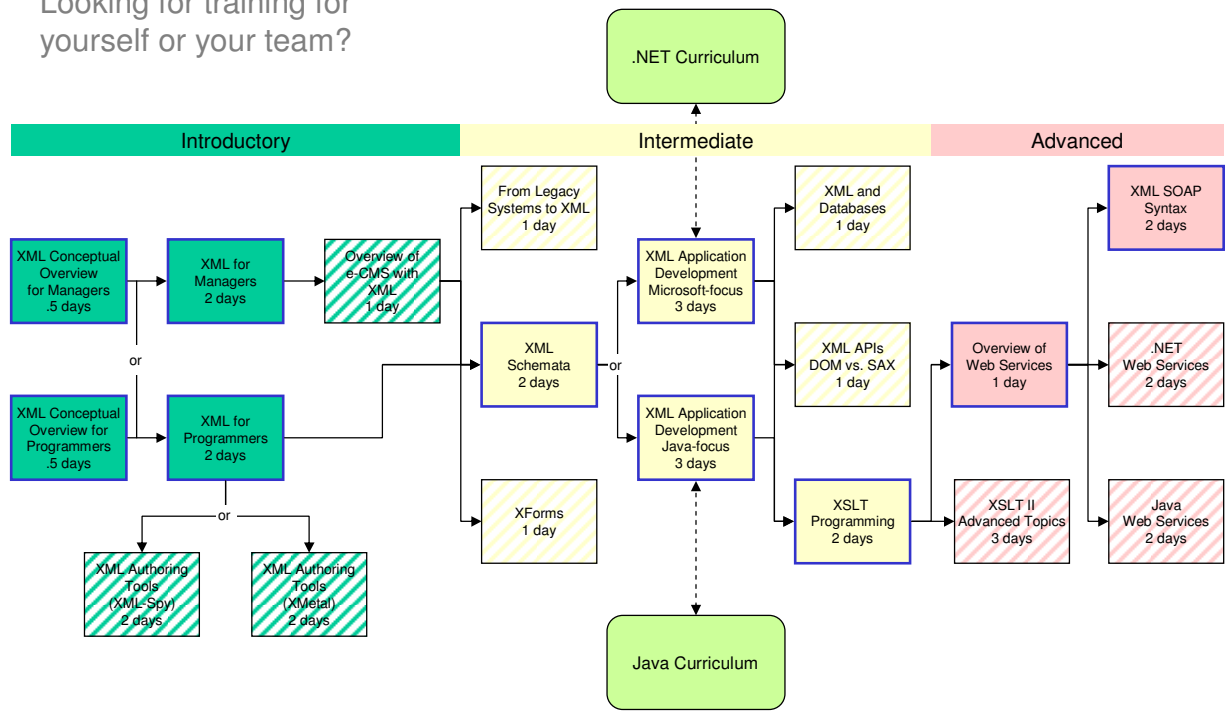
If you have questions after the session is over that you think I might be able to help with, please feel free to contact me by E-Mail. In the event that I don't know the answer to your question, I will try and direct you to a Web or other resource for further information.

Please visit us on the Web at:

http://XMaLpha.com

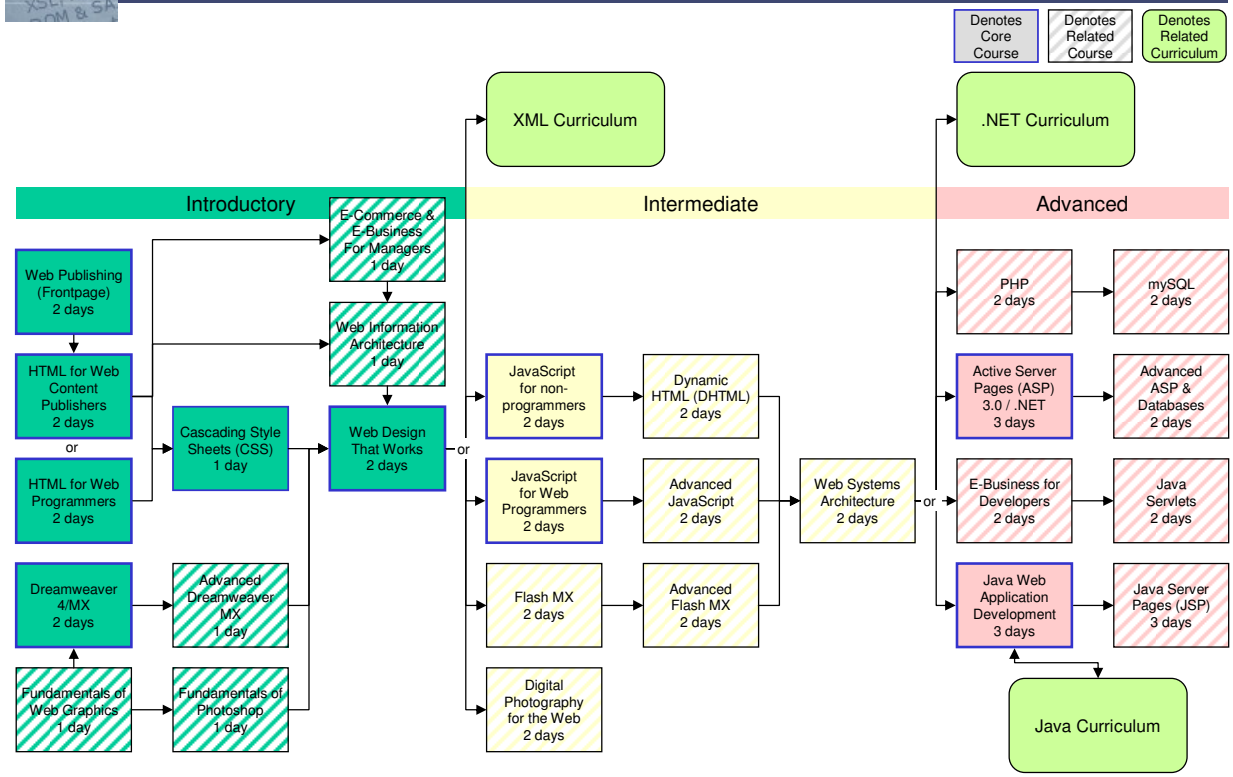
XMaLpha XML Curriculum

Looking for training for yourself or your team?



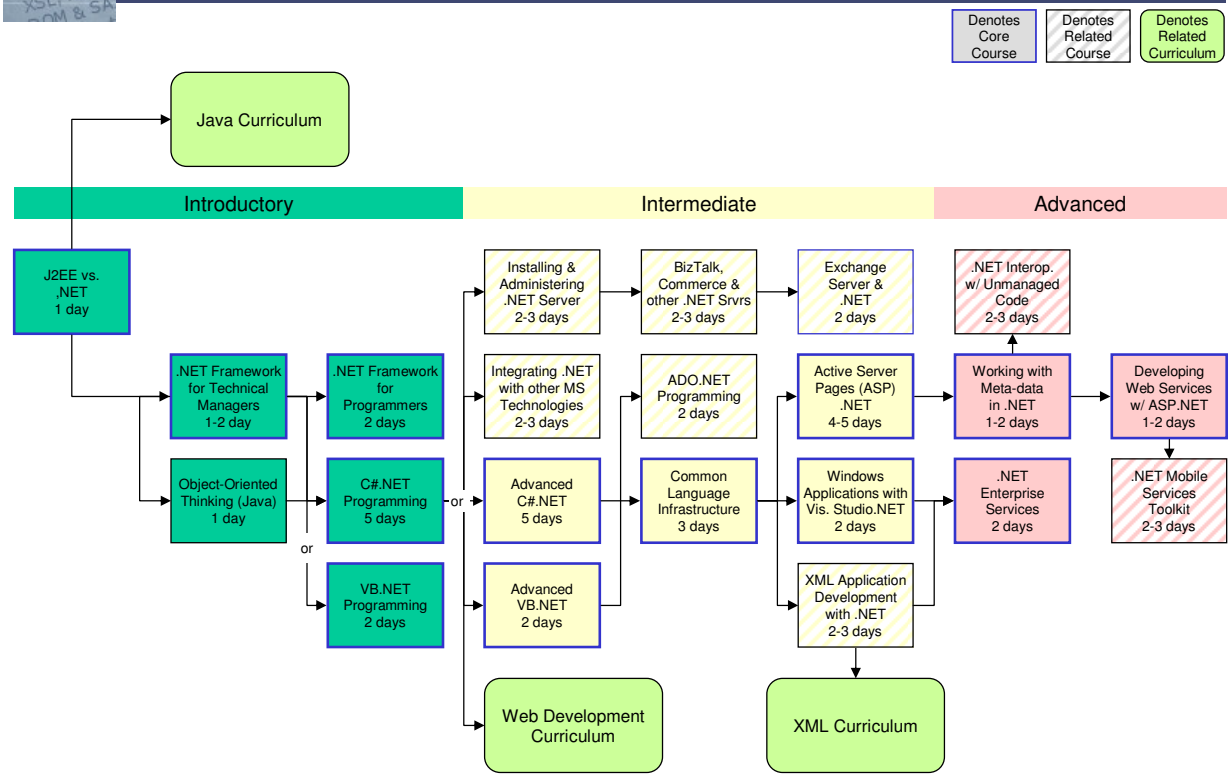
A place for your notes…

XMaLpha Web Development Curriculum



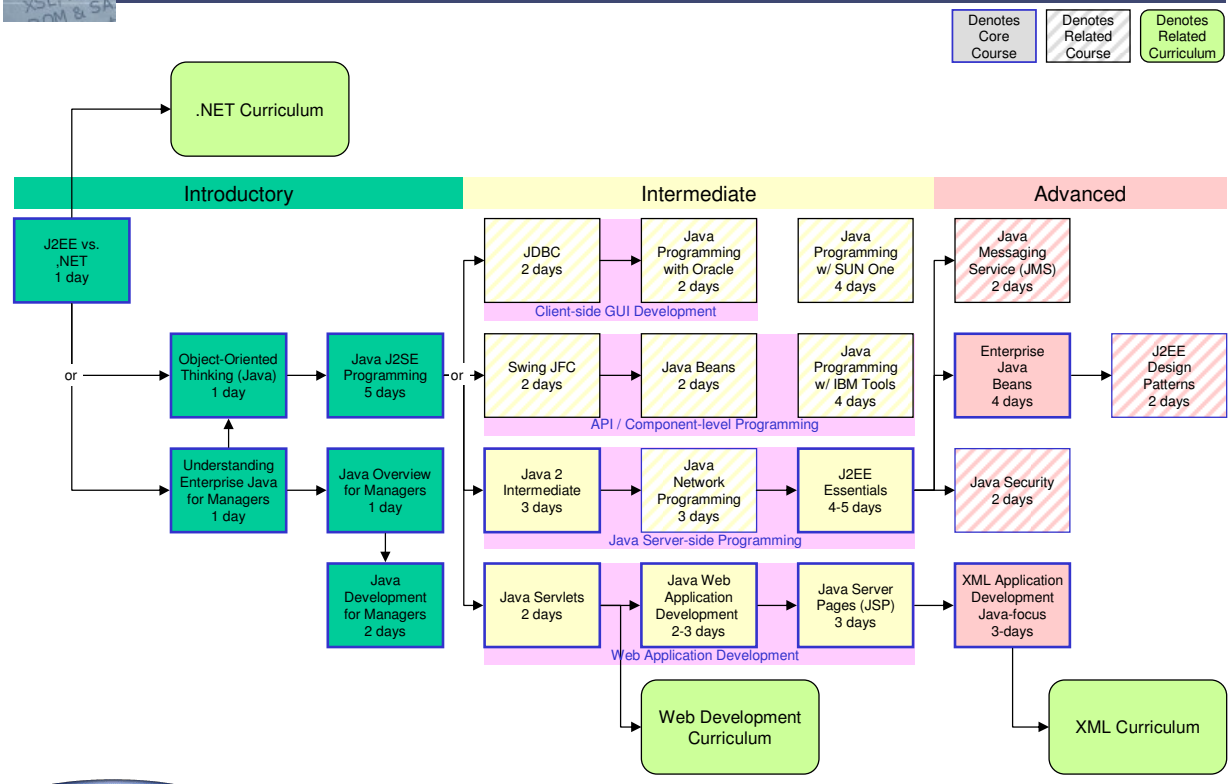
A place for your notes…

XMaLpha .NET Curriculum



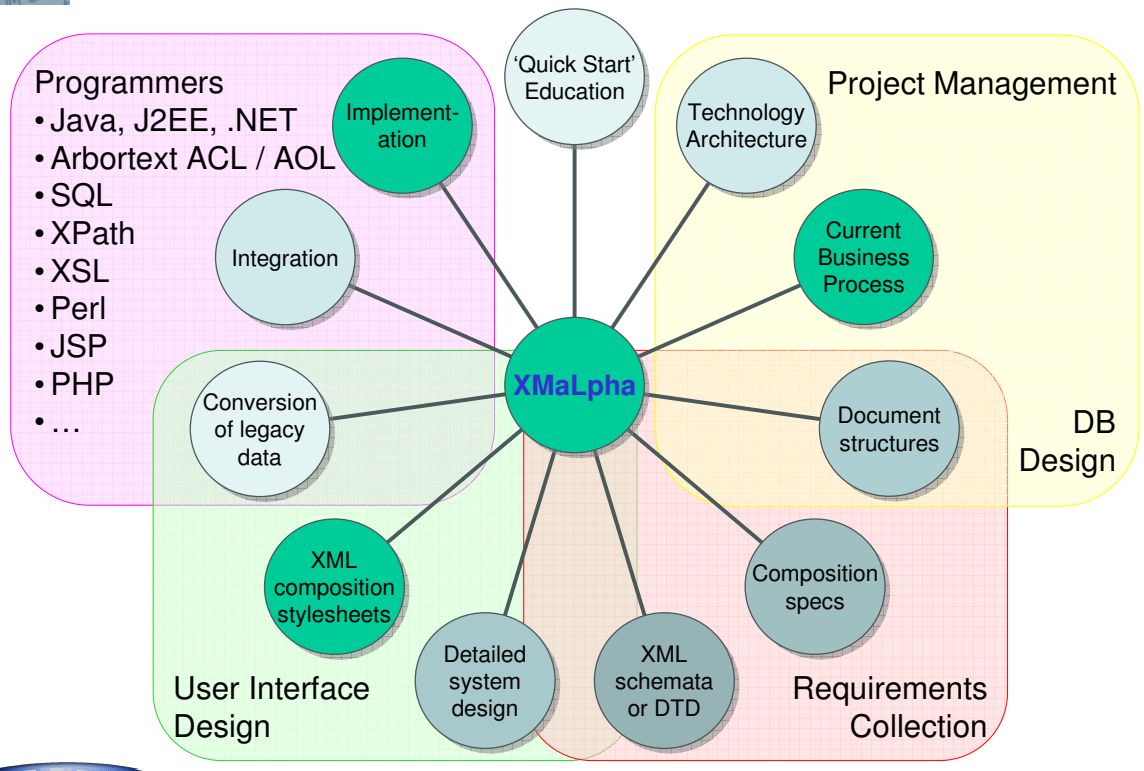
A place for your notes…

XMaLpha Java Curriculum



A place for your notes…

Consulting Services



An independent perspective

A proven track record with extensive XML experience in the legislative environment

XMaLpha Technologies:

- Focus on design and analysis using our experience plus proven techniques
- Build architectural roadmaps and specifications
- Develop the core services using the selected tools and environments
- Design, plan, and deliver integration and implementation