

A photograph of a large iceberg floating in the ocean. The iceberg has several natural arches and openings, creating a series of tunnels. The water is a deep blue, and the sky is a lighter blue. The overall scene is serene and majestic.

# Hierarchy Developer's Guide

**By Project Hierarchy & Unconventional Thinking**  
**Document Version 0.22**

# Table of Contents

*\* If you have this document open in Acrobat Reader, use this table of contents to quickly jump to a page by clicking on the line of the page you'd like to jump to.*

<b>Introduction and Running the Sample Projects .....</b>	<b>3</b>
System Requirements .....	3
Instructions for Running the Samples in Netbeans .....	4
Instructions for Running the Samples in Eclipse .....	7
 <b>Chapter 1 – A Very Quick, Whirlwind Tour of Matrices, Schemas, &amp; Embedded Java Files .....</b>	<b>11</b>
 <b>Chapter 2 – A Tutorial on Creating Matrices and Schemas .....</b>	<b>17</b>
Let's build a matrix .....	17
Field Sets .....	20
Child Descriptors .....	21
The '.matrix' File .....	22
Schemas .....	24
 <b>Chapter 3 - Symbols .....</b>	<b>27</b>
Symbols .....	27
Descriptor Tag-Names – a second type of symbol .....	32
Labels – A Third Type of Symbol .....	33
Summary of How to Use Symbols .....	35
 <b>Chapter 4 - Schemas .....</b>	<b>38</b>
Defining Descriptors .....	38
Defining Field Types .....	40
Using a Schema in a Matrix: Importing the Schema .....	41
Using a Schema in a Matrix: by Specifying the Full Schema Name .....	41
Summary of Usages of the Different Types of Symbols in a Schema .....	43
 <b>Chapter 5 - Matrices .....</b>	<b>45</b>
Matrices, along with Schemas are a new data-type for Java .....	45
The ITEM Descriptor .....	45
 <b>Chapter 6 - Accessing Matrices in your Embedded Java Code .....</b>	<b>48</b>
Main Concepts of Embedded Java Files .....	48
Descriptor Variables .....	50
Multi-Access, Part 1 .....	51
Using a Matrix in an Embedded Java File: Importing the Matrix .....	52
Using a Matrix in an Embedded Java File: by Specifying the Full Matrix-Name .....	53
Annotations .....	54
Multi-Access, Part 2 – Filters .....	59
Matrix access with a dynamically generated label .....	60
Modifying Fields and Descriptors in Matrices .....	61

<b>Chapter 7 – Using the Metacompiler: How to metacompile and debug your code .....</b>	<b>63</b>
The Matrix Metacompilation Process.....	63
The Debugging Process.....	64
Internal Structure of Matrices and Schemas.....	66
Debugging your .schema, .matrix and .mjava Files .....	79
Setting up the Metacompiler.....	82
Dealing with Java Dependencies .....	86
Matrix files that depend on Java files .....	86
Java files that depend on Matrix files .....	89
Java files that both “depend on” and “are depended on” by Matrix files.....	91
<b>Chapter 8 – Starting your own Matrix Project .....</b>	<b>92</b>
Method #1 – Copy One of the Sample Projects .....	92
Method #2 – Starting a New Project using your IDE.....	93
Starting a New Matrix Project in Netbeans – Non-Web Project.....	93
Starting a New Matrix Project in Netbeans – Web Project.....	94
Starting a New Matrix Project in Eclipse – Non-Web Project .....	96
Starting a New Matrix Project in Eclipse – Web Project.....	98
<b>Chapter 9 – Persistent Matrices with “Frictionless Persistence” .....</b>	<b>102</b>
<b>Chapter 10 – Case Studies on When to Use Matrices .....</b>	<b>105</b>
Case #1 – System Configuration File.....	105
Case #2 – N-Dimensional Architecture .....	108
Benefits of N-Dimensional Architecture .....	123
Case #3 – Universal Data Definition.....	125
<b>Final Words .....</b>	<b>129</b>

# Introduction and Running the Sample Projects

In this developer guide, you'll be shown how to use the Hierarchy Metacompiler to create applications using our set of programming-language extensions to the Java language. A description of Hierarchy that seems easiest for devs to understand is that Hierarchy is a compiler for Java that adds something very similar to XML directly into the Java language. We've extended the Java syntax, adding in an XML-like data structure and also new operators used for creating and working with it. But just to make this clear, this new data structure isn't XML, but it has the same hierarchical structure. So, if you're familiar with XML, you should feel right at home with Hierarchy.

Code compiled by Hierarchy is 100% compatible with existing Java. The reason is because Hierarchy is a translator that translates code written in this extended Java syntax into just pure Java. Since the translated code is just pure Java, you'll be able to use all the existing Java libraries you're familiar with. And (not to confuse you), but we don't call Hierarchy a translator or a compiler, we call it a "metacompiler".

The first chapter of this developer guide is a whirlwind tour, giving you a very quick idea of what matrices are. Please start with this chapter as it'll quickly give you a good understanding of how everything works. Chapter 2 is a tutorial on creating schemas and matrices (also, a very important chapter and not to be missed). Chapters 3, 4 and 5 focus on specific aspects of schemas and matrices. And, chapter 6 deals with how to access matrices in special Java files embedded with matrix code.

There are a couple of other chapters later in the guide that we'd like to point out. Chapter 9 talks about persistent matrices. Persistent matrices is a new feature, and by persistence, we don't mean the object-to-database mapping provided by tools like Hibernate. We mean true persistence, where any changes your application makes to any matrices are stored to disk, so the modified matrices are then still available if your application is shutdown and restarted. It's the easiest persistence you'll ever use.

Chapter 8 talks about how to start a new matrix project from scratch. We've saved this for later in the guide, because we encourage you to learn how to work with matrices first by playing with and modifying the sample projects. And then, later, when you have a better understanding of how to program with Hierarchy, you'll be much more ready to learn about setting up your own matrix project.

So, having said that, before we begin, it's highly recommended that you set up the sample project, *MatrixSample\_Basic*, modifying it as you go through this guide, trying out different features. In the zip file that you downloaded, there are a few sample Matrix projects. You can set up the *MatrixSample\_Basic* project in few minutes (very easy to do). And, if you're ready, we'll begin by stepping you through the process.

First, let's go over the system requirements:

## System Requirements

1. Java 1.6 or above installed on your system
2. Ant 1.6 or above installed on your system and can run Ant from the command line.
3. If you'd like to use the sample **web** projects, install Tomcat 6.0 on your system and Java 1.6.
4. A current version of Netbeans (6.9 or higher) or Eclipse on your system
5. Hierarchy's metacompiler works with Win / UNIX / Linux / Mac.

Now, on the next pages, let's meta-compiling and run a sample project. There are two sets of instructions, one for Netbeans and one for Eclipse (The Eclipse instructions follow the ones for Netbeans). Use the correct instructions for your IDE.

## Instructions for Running the Samples in Netbeans

*\* If you're using Eclipse, go to the next set of instructions!*

*\* Also note, don't use these instructions for the PersistenceMatrix\_WebSample! Its instructions can be found in the ReadMe.txt file in the PersistenceMatrix\_WebSample project directory.*

1. After unzipping your Hierarchy download, open up the *MatrixBasic\_Sample* project in the Netbeans IDE (or, any other of the sample projects except the one for Persistence! These instructions should work the same for all to them). The sample projects are located in the `Hierarchy/samples/Netbeans`.
2. If necessary, fix the references to the jars used by the project. You can tell if you need to do this if your project icon shows an error sign. This should mean there are broken references to jars. If you need to fix these, open up the **Properties** for the project, and the **Project Properties** dialog box should pop up. Then, in **Categories** window-pane on the left side, select **Libraries**. In the libraries window-pane, if there are any broken references, remove them and add the corrected references back in. You need references to the following three jars:
  - a. **Hierarchy.jar** – located in `Hierarchy/bin`
  - b. **commons-collections-3.2.1** – located in `Hierarchy/bin/lib`
  - c. **sablecc.jar** – (actually, not needed for most projects) located in `Hierarchy/bin/lib`
  - d. **junit.jar** – you can remove this reference if you'd like, but you can also link to the one provided for you in `Hierarchy/bin/lib`
3. In the base directory of the sample project, find the file called: **hierarchy.properties**. Open this file in Netbeans or a text editor and set the first two properties to the correct directories on your local system:

```
# Set the path to the Hierarchy.jar directory.
hierarchy.jar.dir=/Projects/Hierarchy/Samples/bin

# Set the path to the javac directory.
javac.dir=/Java/jdk1.6.0_11/bin
```

4. If you're working with one of the sample web projects, you should setup tomcat server in your IDE:
  - a. First, install Apache Tomcat *version 6* on your system (it may work with other versions of Tomcat, but the projects have been tested with version 6), if you haven't done so already.
  - b. Next, in Netbeans, you need to add the server to your IDE (again, if you haven't already done so). You do this in the **Tools** menu under **Servers**.
  - c. \* You also need to let the ant build script, `buildJSP.xml`, know where your Tomcat installation is. Open `buildJSP.xml` in Netbeans or in a text editor, and edit the following line to point to your tomcat base dir.

```
<property name="tomcat.home" value="/Java/servers/tomcat-6.0.29"/>
```

5. At this point, you can just use the “clean” and “build” menu items to properly clean and metacompile the sample project. The reason you can do this is because for each of the sample projects, behind the scenes, the Netbeans build script, “build.xml,” has some custom ant code that calls a matrix build-script (“buildMatrix.xml” or “buildJSP.xml”). You can view this custom ant code in your project’s build.xml if you’d like, and while you’re at it, you may also want to take a look at the buildMatrix.xml file too. The custom ant code together with the buildMatrix.xml lets you build all the matrix-related parts of the project directly in Netbeans.

And, since you can build the matrix-related parts of the project directly in Netbeans, you can skip the next steps (steps 5 – 6) which show you how to build these matrix-related parts *manually*. They show you how to do this building of the matrix-related elements outside of Netbeans from the command line. If you’re not interested in learning how this is done, feel free to skip ahead to step 7. ***But, we highly recommended that you try steps 5 and 6!*** The reason is so you can get a more in depth understanding of how the metacompilation process works, which will be useful if you need to debug problems or create your own matrix project.

6. **Now, let’s meta-compiling the code – Optional!** – *As a reminder, the next two steps are not necessary, as the sample project in Netbeans is setup to perform these steps automatically. But, if you’re interested in learning what’s happening behind the scenes, perform these steps.*

If you look at the source code for the sample project, most of it is Hierarchy, meta-code files. You can tell they are meta-code files because they have one of the following extensions: **.matrix**, **.schema**, **.mjava**, or **.mjsp**. We need to meta-compile these into .java files. To do this, on the command line, cd to the sample project’s directory, and run the ant command:

```
ant -f buildMatrix.xml
```

Or, for one of the web projects, run:

```
ant -f buildJSP.xml
```

This should metacompile the project. If you look again in the sample project’s source code, you should see new .java files added into the source. You can open this project up in your IDE to take a look at the generated files.

7. **Next, let’s compile the generated .java files.** At this point, we’ll compile the Java files generated by the Hierarchy metacompiler into byte code. This is done pretty much the standard way. In the IDE and perform a “build”.

*Note: Because of the custom ant code that’s been added to your project’s build.xml, you’ll notice that when you build, it’ll also do the metacompilation again.*

8. **Let’s run the project.** If you’re using Netbeans,
  - a. **For the console sample projects** – like MatrixSample\_Basic, click on the “run project” button. Alternatively, go to the command line, cd to the directory, MatrixSample\_Basic/dist, and run the command:

```
java -jar "MatrixSample_Basic.jar"
```

- b. **For the Web sample projects** – like `MatrixWebSample_Basic`, click on the “run project” button.  
\* In some versions of Netbeans, clicking on Run doesn’t work! When the browser is launched, you’ll get a servlet exception:

```
javax.servlet.ServletException: Wrapper cannot find servlet class  
org.apache.jsp.index_js
```

or

```
Servlet org.apache.jsp.index_jsp is not available
```

If you see this error, try doing a “Deploy” instead of a run. Then, manually opening your browser and hitting: <http://localhost:8080/>

## 9. To clean the project:

- a. **In Netbeans** – by selecting “clean” from the project menu, this will both clean the .java files generated by the meta-compiler, and the .class files generated by the java compiler. Behind the scenes, the project’s Netbeans “build.xml” ant script calls the Hierarchy “buildMatrix.xml” ant script to clean out the generated Java files (see the next step to see exactly what in this script is called).
- b. **OR, to clean the matrix-related parts of the project from the command line** – In a terminal window, at the project’s directory, type:

```
ant -f buildMatrix.xml clean
```

Or, for a web project:

```
ant -f buildJSP.xml clean
```

## 10. If you run into any problems, please refer to the FAQ document! It contains answers to common problems.

## 11. Also, if you want to try some the examples in this Developer Guide for yourself, you may want to work with a copy one of the sample projects.

- Copy one of the sample projects into its own directory.
- Modify the properties in **hierarchy.properties** to point to the correct paths in your system
- If you’re copying one of the web projects, modify the `buildJSP.xml` file to point `tomcat.home` to the correct file path of your tomcat installation.
- Try metacompiling and then compiling the project to see if it works.
- Now, you can try out any modifications you’d like.

... And, just as a reminder, you can metacompile and properly clean the sample projects directly from the Netbeans IDE (meaning, you don’t have to do steps 6-7 in the instructions). Also, later, after reading through more the guide and when you feel comfortable programming in Hierarchy, feel free to jump to chapter 8, which explains how to setup your own Hierarchy projects from scratch.

Enjoy!

## Instructions for Running the Samples in Eclipse

*\*Note that these instructions work for all the Sample projects, not just the MatrixBasic\_Sample.*

*\*Also note that, there is a sample project that shows how Persistence with matrices work. It's pretty neat! But unfortunately, this sample project does not currently work in Eclipse, only in Netbeans. If you'd like to try it, you'll have to use the Netbeans IDE.*

1. After unzipping your Hierarchy download, open up all the sample projects in Eclipse. The sample projects are located in the `Hierarchy/samples/Eclipse` directory. To open the projects:
  - a. Right click in the package explorer window and select **"import"** from the pop up menu. Then, from the context menu, select, "import".
  - b. Then, select "Existing Projects into Workspace"
  - c. Now, browse to the `eclipse` folder in the samples directory of the Hierarchy Project. This will let you import *all* the sample projects in at the same time!
  - d. Click ok
2. In the *MatrixBasic\_Sample* project (or whichever sample project you're using), you'll have to fix the references to the jars used by the project. First, switch to the *Java EE* perspective in Eclipse. Then, you can confirm you need to do fix the libraries by looking at the project's build errors in the **Markers** window pane. These errors should say that there are broken references to jars. To fix these, open up the properties dialog for the project by right clicking on the project in the project explorer and in the context menu that pops up, select **Properties**. Then in this dialog box, select the **Java Build Path** settings on the left side nav. Then, select the **Libraries** tab. In the libraries tab, there should be some broken references to jars, edit them to point to the correct jars. You need references to the following three jars:
  - a. **Hierarchy.jar** – located in `Hierarchy/bin`
  - b. **commons-collections-3.2.1** – located in `Hierarchy/bin/lib`
  - c. **sablecc.jar** – (actually, not needed for most projects) located in `Hierarchy/bin/lib`
  - d. **junit.jar** – you can remove this reference if you'd like as it's not going to be used, but you can also link to the one provided for you in `Hierarchy/bin/lib`
3. In Eclipse, turn off "Build Automatically." Because the metacompilation process for these projects is more complicated than just the standard, Java build-process, you probably don't want to build every time you save. So, you can turn off build auto by unchecking the menu item, **Projects->Build Automatically**.
4. In the directory for the sample project, find the file called: **hierarchy.properties**. Open this file in Eclipse or in a text editor and set the first two properties to the correct directories on your local system:

```
# Set the path to the Hierarchy.jar directory.
hierarchy.jar.dir=/Projects/Hierarchy/Samples/bin

# Set the path to the javac directory.
javac.dir=/Java/jdk1.6.0_11/bin
```



5. If you're working with one of the sample web projects, you should setup tomcat server in your IDE:
  - a. First, install Apache Tomcat *version 6* on your system (it may work with other versions of Tomcat, but the projects have been tested with version 6), if you haven't done so already.
  - b. In Eclipse, you add the server to the IDE in the **Servers** window (usually, one of the small tabbed windows near the bottom of the IDE). Right clicking in the server window, then click on **New > Server**. Next, use the wizard to add the server to the IDE.
  - c. \* You also need to let the ant build script, `buildJSP.xml`, know where your Tomcat installation is. Open `buildJSP.xml` in Eclipse or in a text editor, and edit the following line to point to your tomcat base dir.

```
<property name="tomcat.home" value="/Java/servers/tomcat-6.0.29"/>
```

6. You may have to fix your project's JRE settings, especially for the web projects. In your "Markers" window pane, if you see the error:

```
Java Build Path Problems
  x Unbound classpath container: ` JRE System Library [jdk1.6.0_21]
    in project ...
```

This means your JRE settings for some of your sample projects are incorrect. To fix the JRE problem:

- a. Open your project's Properties dialog.
  - b. In the left nav, select "Java Build Path."
  - c. Click on the "Libraries" tab.
  - d. You should see a red 'x' by the library, "JRE System Library." Select this entry and then click the Edit button.
  - e. A dialog will pop up giving you three options for selecting a JRE. It doesn't matter which one you chose, as long as the JRE version is 6, and that it resolves correctly!
  - f. Check that this worked. In the Markers pane, the error should have gone away for this project – note that you may have more than one sample project with this same error, so even if this error still shows up after the fix, double check to make sure that the one for the project you just fixed has gone away.
7. At this point, you can just use the "clean" and "build" menu items to properly clean and metacompile the sample project. The reason you can do this is because for each of the sample projects, behind the scenes, your Eclipse project is using "project builders" to call some custom ant code in a matrix build-script ("buildMatrix.xml" or "buildJSP.xml"). If you'd like, you can view your project's builders in the project's *properties* dialog, under *Builders* (it's the builder called "Build MatrixSample\_Basic"). And, while you're at it, you may also want to take a look at the `buildMatrix.xml` file too. The builder together with the `buildMatrix.xml` lets you build all the matrix-related parts of the project directly in Eclipse.

And, since you can build the matrix-related parts of the project directly in Eclipse, you can skip the next steps (steps 8 – 10) which show you how to build these matrix-related parts *manually*. They show you how to do this building of the matrix-related elements outside of Eclipse from the command line. If you're not interested in learning how this is done, feel free to skip ahead to step 11. **But, we highly recommended that you try steps 8 through 10!** The reason is so you can get a more in depth understanding of how the metacompilation process works, which will be useful if you need to debug problems or create your own matrix project.

8. **Now, let's meta-compile the code. – Optional!** – As a reminder, the next three steps are not necessary, as the sample project in Eclipse is setup to perform these steps automatically. But, if you're interested in learning what's happening behind the scenes, perform these steps.

If you look at the source code for the sample project, most of it is Hierarchy, meta-code files. You can tell they are meta-code files because they have one of the following extensions: **.matrix**, **.schema**, **.mjava**, or **.mjsp**. We need to meta-compile these into .java files. To do this, on the command line, go to the sample project's directory, and run the ant command:

```
ant -f buildMatrix.xml
```

Or, for one of the web projects, run:

```
ant -f buildJSP.xml
```

This should metacompile the project. If you look again in the sample project's source code, you should see new .java files added into the source.

9. **In Eclipse, for the generated .java files to show up in the project, you need to “refresh” the project.** You can do this in the package explorer window by right clicking on the project and selecting “refresh” from the pop up menu. ***This is an important step, so don't forget to do this every time after you metacompile from the command line!***
10. **Next, let's compile the generated .java files.** At this point, we'll compile the Java files generated by the Hierarchy metacompiler into byte code. This is done pretty much the standard way. In the IDE and perform a “build.”

*Note: Because of the project's Builders, you'll notice that when you build, it'll also do the metacompilation again.*

11. **Let's run the project.** In Eclipse, right click on the project and select “run as” from the pop up menu.

**For the Non-Web Projects:**

- Select “Java Application”
- Then, select “SimpleEmbeddedFile”

Alternatively, go to the command line, cd to the directory, `MatrixSample_Basic/dist`, and run the command:

```
java -jar "MatrixSample_Basic.jar"
```

**For the Web Projects:**

- Select “Run on Server”
- “Tomcat v6.0 Server”
- Click on Next
- Add the sample web project.
- Click Finish

## 12. To clean the project:

In a terminal window, at the project's directory, type:

```
ant -f buildMatrix.xml clean
```

Or, for a web project:

```
ant -f buildJSP.xml clean
```

## 13. If you run into any problems, please refer to the FAQ document! It contains answers to common problems.

## 14. Also, if you want to try some the examples in this Developer Guide for yourself, you may want to work with a copy one of the sample projects.

- a. Copy one of the sample projects into its own directory.
- b. Modify the properties in **hierarchy.properties** to point to the correct paths in your system
- c. If you're copying one of the web projects, modify the buildJSP.xml file to point `tomcat.home` to the correct file path of your tomcat installation.
- d. Try metacompiling and then compiling the project to see if it works.
- e. Now, you can try out any modifications you'd like.

... That's it for setting up the samples in Eclipse. And, after reading through more of the guide and when you feel comfortable programming in Hierarchy, feel free to jump to chapter 8, which explains how to setup your own Hierarchy projects from scratch.

Enjoy!

# Chapter 1 – A Very Quick, Whirlwind Tour of Matrices, Schemas, & Embedded Java Files

To start off, let's take a whirlwind tour of Matrices, Schemas, and Embedded Java files. Matrices have many concepts and syntax that parallel XML, so a very brief tour is probably all that's needed to give you a good, high-level understanding of how matrices work. If you're already read through the "Learn More" section in the website, you may just want to skip ahead to chapter 2, because this is exactly the same information as what's been included on the site.

*Hierarchy adds the 'missing data-type' to Java...*

A description of Hierarchy that seems to give devs a quick, intuitive understanding of it is that Hierarchy is a compiler for Java that adds something very similar to XML directly into the Java language. We've extended the Java syntax, adding in an XML-like data-type and also new operators used for creating and working with it. But just to make this clear, this new data type isn't XML, but it has the same hierarchical structure. So, if you're familiar with XML, you should feel right at home with Hierarchy.

In fact, Hierarchy adds three, new file-types to Java, all of which map to equivalent file types in XML. Hierarchy adds:

- **.matrix file** - which is equivalent to the .xml, and holds hierarchical data
- **.schema file** - which is equivalent to an .xsd, and has the schema information for describing the structure of the data.
- **.mjava file** - which is equivalent to the .xsl file. This one isn't an exact match. A .xsl is an entire language for processing xml data, while the .mjava file is just a regular .java file with new instructions for accessing the hierarchical data in the .matrix files.

It's probably easier to show you how it works than to continue to try explain it. So, let's dive right in and take a look at an example. Let's say we're building a website for a pet store. Specifically, we'll build the news section for the site's homepage. Here's what the web page looks like:

## Willie's Pets and Things

### *news*

#### **Adopt-a-Pet Day is this Sunday!**

*March 17<sup>th</sup>*

Your chance to adopt a lonely doggie is right around the corner. Come by Willie's Pets on Sunday to find the personality that's a perfect match for you!

#### **20% off Fanciful Dog Spread**

*March 12<sup>th</sup>*

For one day only, come and pick up your tub of Fanciful Dog Spread! A truly refined and tasteful topping your dog will find dreamy.

Now, as we mentioned, we're just going to focus on the *news* section of this page. First, let's create a file with all the hierarchical data. As we talked about earlier, this type of data is stored in a .matrix file (which is equivalent to the .xml). Our matrix file will contain all the news story data for this page:

```

package com.williespetstore;

import java.util.Date;
import java.text.DateFormat;

MATRIX WilliesPetstore.Content USES (News.Schema) {

    NEWS: {"HomePage News Blurbs"} {

        NEWS.STORY +`Adopt a Pet`: {
            DateFormat.getDateInstance().parse("March 17, 2010"),
            "Adopt a Pet",
            "Your chance to adopt a lonely doggie is right " +
            "around the corner. Come by Willie's Pets on Sunday " +
            "to find the personality that's your perfect match!"
        };

        NEWS.STORY +`Fanciful Dog Food`: {
            DateFormat.getDateInstance().parse("March 10, 2010"),
            "20% off Fanciful Dog Spread",
            "For one day only, come and pick up your tub of " +
            "Fanciful Dog Spread! A truly refined and tasteful " +
            "topping your dog will dream about."
        };
    }
}

```

*The Willie's PetStore matrix file: WilliesPetstore\$\_\_Content.matrix*

As you can probably see, the .matrix file has a very similar structure to XML. It has a root node called `WilliesPetstore.Content`, which has a child, `NEWS` node. And then, this `NEWS` node itself is a parent to multiple, child `NEWS.STORY` nodes. Now, in the `NEWS.STORY` nodes, this is where we're storing the actual values, which are held in different fields. For instance, in the first `NEWS.STORY` node, the strings, "Adopt a Pet" and "Your chance to adopt a lonely doggie..." are both values stored in *fields*. 'Fields' in Hierarchy are equivalent to 'attributes' in XML.

Conceptually, you should think of a Matrix as a new data type for Java. It's on the same level as a regular class. In fact, matrices aren't static like XML documents, but, instead like instances of a Java class, are mutable. You can dynamically add, remove, and modify values at runtime. You can even add whole new subtrees. We created Hierarchy with the idea of building on what's already in Java, and matrices were designed to naturally fit in as a new, third data-type. In between primitive types (like int's) and classes, matrices are a third, new data-type available for you to use.

And briefly, to finish this conceptual picture of matrices, behind the scenes, a matrix is just a collection of regular, Java objects. This collection has Descriptor objects, which have FieldSet objects, and to wrap the whole thing is a Matrix object.

The next file type is the .schema file. Let's define the schema for the news story matrix. We'll create a new file named, `News$__Schema.schema`, and add the following code:

```

package com.williespetstore;

SCHEMA News.Schema {

    DESCRIPTOR +:%NEWS {
        FIELD.NAMES: { +:%NewsSectionName };
        FIELD.TYPES: { :String };

        DESCRIPTOR +:%NEWS.STORY {

            FIELD.NAMES: { +:%StoryDate, +:%Title, +:%StoryContent };
            FIELD.DESC: { "The date of the story",
                "The title of the story",
                "The content of the story" };
            FIELD.TYPES: { +:"java.util.Date", :String, :String };

        }
    }
}

```

*The News Story schema file: News\_\_\$Schema.schema*


As you can see, we have the definition for a parent, NEWS node (or, in our terminology, we call nodes descriptors). Off this NEWS descriptor, we define a child descriptor, NEWS.STORY. This means in a matrix that uses this schema, a parent, NEWS descriptor can have many child, NEWS.STORY descriptors. The NEWS.STORY descriptors would contain the actual text of each of the different stories.

The one part that confuses developers the most in schemas is how to define the fields for a descriptor. You do this with the FIELD.NAMES, FIELD.TYPES and FIELD.DESC descriptors. But, before we talk about this, we need to look back, and take another look at the matrix example. In the WilliesPetstore.Content matrix, you may notice that for each of the field values, matrices have no field names!

```

NEWS.STORY +`Adopt a Pet`: {
    "March 17, 2010", "Adopt a Pet", "Your chance to adopt a doggie ..."
};

```


 In Matrices, there are no field names like xml! (In xml, these fields would look something like: date="March 17, 2010" title="Adopt a Pet"... )

The reason there are no field names is because for a value, the field it belongs to is determined by its position! So, for the NEWS.STORY descriptor, the first value in the fieldset is always going to be the date, the second is going to be the title, and the third is going to be the storyContent. And, the place where you set this relationship up is in the schema. Here, you can see this relationship below:

In the Matrix, here's a NEWS.STORY descriptor:

```
NEWS.STORY +`Adopt a Pet`: { "3/17/2010", "Adopt a Pet", "Lots of text" };
```

Descriptor Tag  
Name Definition

Story Date  
Field Definition

Title  
Field Definition

Story Date  
Field Definition

Here's its definition  
in the Schema:

```
DESCRIPTOR +%NEWS.STORY {  
    FIELD.NAMES: {   +:%StoryDate,      +:%Title,      +:%StoryContent };  
    FIELD.DESC: {    "The date",        "The title",   "The content" };  
    FIELD.TYPES: {   +:"java.util.Date", :String,       :String };  
    FIELD.DEFAULTS: { null,             null,          null };  
}
```

As you can see above, in a matrix, the position of the value determines which field it belongs to. And position is also used in the schema, in the field's definition. So, for the first field, StoryDate, all its field information is defined in the first position of the FIELD descriptors! And, for the second field, Title, all its field information is defined in second position, and so on... You probably got the hang of this, but we'll revisit this in more detail in the next chapter.

Note thought that Matrices do still allow you to do the `fieldName='fieldValue'` style of mapping, but we actually found that this extra text tends to make your code messy, so when we can, we use the syntax without the field names.

Compared to XML schemas, the structure of matrix schemas should be easier to follow. By just looking at the schema file and comparing it with the matrix file, you should easily see that the structure of the Descriptor definitions in the schema correspond to the structure of the descriptors in the matrix. If you haven't done so already, please take a quick look back at the matrix file and try to see where the descriptor definitions in the schema line up with the actual descriptors in the matrix.

So now, as a last step, let's create a console application that accesses the matrix and prints out the matrix's content in an embedded java file. This console app outputs the home page to standard out (of course, for a web page, we'd normally use a JSP or JSF file, but for the sake of simplicity, we'll output the page to the console). To create this app, we'll use the last file type, the .mjava file:

```

package com.williespetstore;

import MATRIX com.williespetstore::WilliesPetstore.Content;

public class WilliesPetStoreConsoleApp {

    public static void main(String[] args) {

        System.out.println("Willie's Pets and Things");
        System.out.println("");
        System.out.println("news:");

        for(Descriptor<WilliesPetstore.Content->NEWS->NEWS.STORY> newsStoryDesc
            : WilliesPetstore.Content->NEWS->NEWS.STORY{*}) {

            System.out.println(newsStoryDesc:>Title);
            System.out.println("_____");
            System.out.println( (newsStoryDesc:>StoryDate).toString() );
            System.out.println(newsStoryDesc:>StoryContent + "\n");

        }

        ANNOTATIONS {
            DEFAULT { return null; }
        }
    }
}

```

*Embedded Java file for Willie's Pet-Store console-app:  
WilliesPetStoreConsoleApp.mjava*

These "embedded" Java files are basically the same as regular a Java file except they have some new instructions. Notice how this file really is just a regular Java class, except for some extra instructions we've added for accessing the matrices. These new, matrix instructions are all in bold. Embedded Java files end with a .mjava file extension.

And, just to finish this sample, let's try and run it. Here's the console output from this application:

```

Willie's Pets and Things

news:
Adopt a Pet

-----
Wed Mar 17 00:00:00 PDT 2010
Your chance to adopt a lonely doggie is right around the corner.

20% off Fanciful Dog Spread

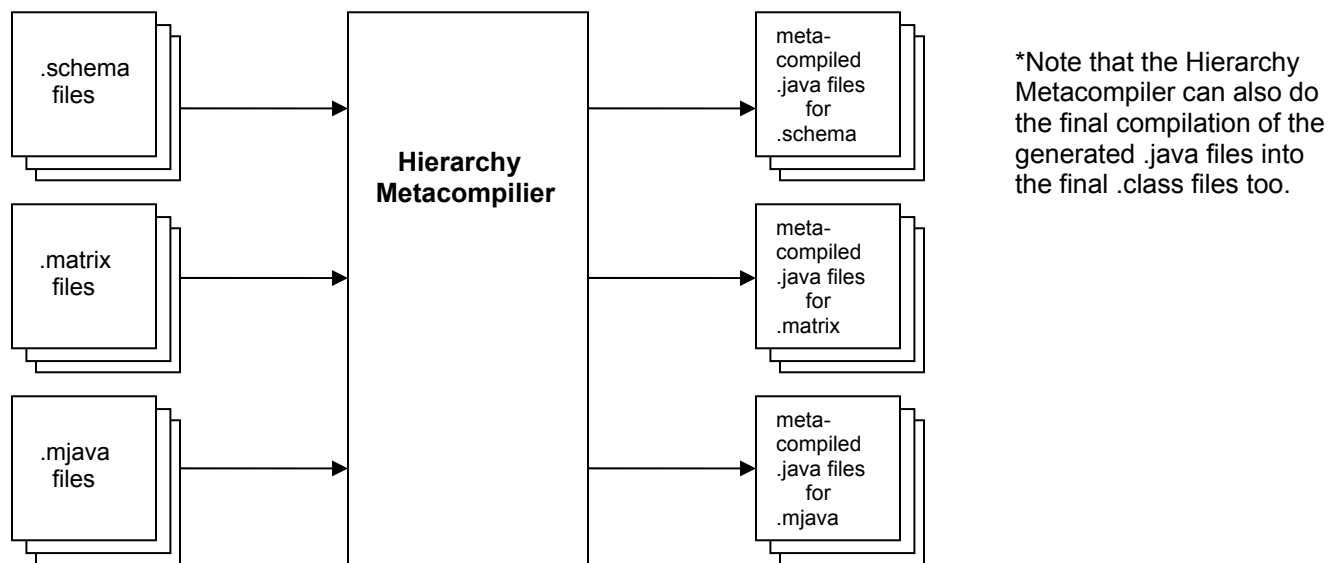
-----
Wed Mar 10 00:00:00 PST 2010
For one day only, come and pick up your tub of Fanciful Dog Spread!

```

At this point, you may be wondering how we provide these language extensions to Java. The way we do this is through "translation," similar to the translation that the Java compiler does in converting Java code into byte code. But with Hierarchy, it translates (or "meta-compiles") code written in our extended, Java syntax with its new, hierarchical data-structure into just pure Java. And, because this meta-compiled code is just Java, you'll



be able to use all the class libraries you're familiar with. Here's a diagram that shows what goes into the metacompiler and what comes out:



The main idea to take away from this diagram is that the Hierarchy Metacompiler takes as input `.schema`, `.matrix`, and `.mjava` files, and outputs pure `.java` versions. These java versions of the files can now be used like regular Java files because they are just plain old, regular Java (well, except they have about three dependencies on jar files that must be included). Also, the nice thing is that the meta-compiled code that Hierarchy generates is 100% compatible with existing Java. Since the translated code is just pure Java, you'll be able to use all the existing Java libraries you're familiar with.

That's it for this quick overview! Hope you found it easy to learn the fundamentals of Hierarchy - we worked really hard to balance simplicity with a rich set of features. The next chapter starts the tutorial, where we'll dive deeper into each of the topics we briefly mentioned in the tour.

## Chapter 2 – A Tutorial on Creating Matrices and Schemas

### Let's build a matrix

Let's take the example from the previous section in the whirlwind tour and go into more depth. In the quick tour, we briefly showed you what matrices and schemas look like. Now, we'll take a deeper look at each of these. So, from the tour, here's the web page we created:

#### Willie's Pets and Things

*news*

##### **Adopt-a-Pet Day is this Sunday!**

*March 17<sup>th</sup>*

Your chance to adopt a lonely doggie is right around the corner. Come by Willie's Pets on Sunday to find the personality that's a perfect match for you!

##### **20% off Fanciful Dog Spread**

*March 12<sup>th</sup>*

For one day only, come and pick up your tub of Fanciful Dog Spread! A truly refined and tasteful topping your dog will find dreamy.

#### ***Example 1***

Let's also revisit the matrix for this page. Actually, in Matrix Programming, you'd typically need to create a schema first before you created the matrix, but, for the sake of ease of learning, we're going to start by going over the matrix first. So, here's what our news matrix would look like for the pet store's new blurbs:

```

package com.williespetstore;

import java.util.Date;
import java.text.DateFormat;

MATRIX WilliesPetstore.Content USES (News.Schema) {

    NEWS: {"HomePage News Blurbs"} {

        NEWS.STORY +`Adopt a Pet`: {
            DateFormat.getDateInstance().parse("March 17, 2010"), "Adopt a Pet",
            "Your chance to adopt a lonely doggie is right around the " +
            "corner. Come by Willie's Pets on Sunday to find " +
            "the personality that's a perfect match for you!"
        };

        NEWS.STORY +`Fanciful Dog Food`: {
            DateFormat.getDateInstance().parse("March 10, 2010"),
            "20% off Fanciful Dog Spread",
            "For one day only, come and pick up your tub of Fanciful Dog " +
            "Spread! A truly refined and tasteful topping your dog will " +
            "dream about."
        };
    }
}

```

### Example 2

As we talked about in the tour, matrices look a lot like XML in their structure, and you may already be able to basically follow how they work. Now, to go in more depth, one major difference between XML and Matrices is how attributes work. In XML, you define tag attributes using: `<attribute name> = <attribute value>`. In matrices, you know what field a value belongs to by what position it is in the “field set”:

#### XML version:

```
<news-story date="03/17/2010" title="Adopt a Pet" content="..." />
```

#### FieldSet

#### Matrix version:

```

NEWS.STORY +`Adopt a Pet`: {
    DateFormat.getDateInstance().parse("March 10, 2010"),
    "Adopt a Pet", "lots of text"
};

```

### Example 3

As you see in the example, the first field in the matrix fieldset holds the story date, the next holds the story title, and the last holds the text. Remember, unlike xml, the field name is left out, and it's the position of a field value in a field set determines which field it belongs to (we'll discuss this in more detail shortly).

In xml, `<news-story .../>` is called a tag. In Matrix Programming, the equivalent:

`NEWS.STORY +`Adopt a Pet`: {...}`, is called a **Descriptor**. Now, let's go over the different parts of a descriptor. There are four parts: the descriptor tag-name, the label, the field set, and the child-descriptor set. These are pointed out to you in the example below:

Descriptor Tag-Name	Label	Field Set
↓	↓	↓
<pre>NEWS.STORY +`Adopt a Pet`: { "03/17", "Adopt a Pet", "lots of text here"} {</pre>		
<pre>    // Child descriptors go inside here</pre>		
<pre>}</pre>		

#### Example 4

And, remember, as we mentioned in the tour, matrices aren't static, like XML, but instead, think of them like a new data-type for Java. Instances of matrices are actually dynamic, and can be modified at runtime. More on this later. For now, let's get back to discussing each part of a matrices in more detail. First, we'll discuss the descriptor tag-name.

### Descriptor Tag-Names

The only part of a descriptor that is required is the descriptor tag-name, the rest of the elements are optional. The descriptor tag-name is equivalent to the tag name in XML. In XML the tag name describes the type of tag, and in Matrices, the descriptor tag-name describes the type of descriptor. And, just like with XML, you can have multiple child nodes with the same descriptor tag-name, meaning each child descriptor does not have to be unique.

```
NEWS: {"HomePage News Blurbs"} {
    NEWS.STORY: { "03/17/2010", "Adopt a Pet", "lots of text"};
    NEWS.STORY: { "03/10/2010", "Fanciful Dog Food", "lots of text"};
}
```



#### Example 5

*\*Note: From example to example, the running sample maybe changed around to illustrate different points.*


Here, there are two, non-unique, NEWS.STORY child-descriptors. This is perfectly valid syntax.

### Labels

The label is an optional **unique** identifier for a descriptor. Among a set of child descriptors, you uniquely identify a descriptor by adding a label right after the descriptor tag name. For example:


```
NEWS: {
    NEWS.STORY +`News story 1`: { "03/18/2010", "News Story # 1", "text 1"};
    NEWS.STORY +`News story 2`: { "03/17/2010", "News Story # 2", "text 2"};
}
```

As you can see from the example, a label is defined using back ticks, ``My Label``. Again, labels are optional, and do not need to be added to a descriptor (see Example 5). *But*, once you do add a label, it *must* be unique among all the other child descriptors for a parent. The label does not need to be unique for the entire matrix, just unique among all the child nodes for a parent. So the following is bad matrix syntax:



```
NEWS: {
  NEWS.STORY +`My Label`: { "03/17/2010", "Adopt a Pet", "text"};
  OTHER.DESC +`My Label`: { 123456 }; // BAD! - they cannot have the same
    // label (even if they have different descriptor tag names)
}
```

But, this next example is good, because the two descriptors with the same label do NOT have the same parent descriptor!



```
NEWS: {"HomePage News Blurbs"} {
  NEWS.STORY +`Adopt a Pet`: { "03/17/10", "Adopt a Pet", "text"};
}
NEWS: { "Internal Company News Blurbs" } {
  NEWS.STORY +`Adopt a Pet`: { "03/17/10", "Mandatory Pet adoption",
    "text"}; // GOOD! Different parent descriptors!
}
```

*Example 6*

Remember, labels are optional, so there is no uniqueness constraint on child nodes with no labels. That's why the *Example 5* is also valid syntax.

## Field Sets

A field set holds values for the descriptors, such as ints, longs, chars and other Java primitive types. Field sets can also hold Java object types and even have Java expressions whose result is a primitive or an object. Again, comparing Matrices to XML, a fieldset is equivalent to the attributes of a XML tag (see *Example 3*). The way you indicate that a descriptor has a fieldset is by including a colon after the label (or the descriptor tag-name if there is no label), followed by curly braces which hold your field set values:

<b>Colon indicates a Field Set</b> ↓	<b>This field value is a Java expression whose result is a Java Object</b> ↓
---	---

```
NEWS.STORY +`Adopt a Pet`: { DateFormat.getDateInstance().parse("March 17, 2010"),
    "Adopt a Pet", "lots of text" };

// Example of a descriptor with a field set, but WITH OUT a label
NEWS.STORY: { DateFormat.getDateInstance().parse("March 17, 2010"),
    "News Story with out a label", "lots of text" };
```

*Example 7*

In a Matrix, an attribute is called a field which has a field value. And, as previously mentioned earlier on, you don't have to specify the name of a field like you do in XML. You know which field a field value belongs to by its position in the field set (this position is defined in the schema file, which we'll get to shortly). Also, there actually is a syntax to specify the field name for each field value (something similar to XML's attributes), but we'll get to that later.

If you don't need a field set, then simply leave off the colon and the curly braces. For instance, we could redo the running example so that the outer descriptor, `NEWS`, could have no fieldset. It would look like this:

```
NEWS {  
  NEWS.STORY: { "03/17/2010", "Adopt a Pet", "lots of text"};  
}
```

*Example 8 – no field set on the outer NEWS descriptor*

We'll come back to fieldsets when we go over schemas. For now, let's move on to child descriptors.

## Child Descriptors

Like in XML, a descriptor node can have child nodes. This is done by including a set of curly braces *with no colon* at the end of the descriptor.

### Child Descriptor Set

```
NEWS: { "HomePage News Blurbs" } {  
  NEWS.STORY: { "03/17/2010", "Adopt a Pet", "lots of text"};  
  NEWS.STORY: { "03/10/2010", "Fanciful Dog Food", "lots of text"};  
}  
↑
```

*Example 9*

**\* Notice, for a child descriptor set, there is no colon right before the curly braces!!**

The purpose of child descriptors in matrices is exactly the same as the purpose of child nodes in xml, parent descriptors "have" or contain child descriptors.

In the previous example, we a descriptor with a fieldset with a child descriptor set. But, if you want to add a child descriptor set but with *no* fieldset, leave off fieldset's "colon with curly braces" block:

### No fieldset

```
NEWS {  
  NEWS.STORY: { "03/17/2010", "Adopt a Pet", "lots of text"};  
  NEWS.STORY: { "03/10/2010", "Fanciful Dog Food", "lots of text"};  
}
```

*Example 10 – no field set*

*But*, if your descriptor has *no child descriptor set*, you can't simply leave off the child descriptor set's curly-braced block. In this situation, you need to replace the missing curly braces and end the descriptor with a semi-colon (as you may have noticed already in many of our examples):

**Semi-colon indicates no child-descriptors and also the end of descriptor**

NEWS.STORY: { "03/17/2010", "Adopt a Pet", "lots of text"}; 😊

// Here's an example with no field set and no child descriptors 😊  
NEWS.STORY;

// But, a descriptor with no child descriptors & no semi-colon is bad! 😞  
NEWS.STORY: { "03/17/2010", "Adopt a Pet", "lots of text"}

↑  
**Needs a semi-colon!**

*Example 11 – no child descriptors*

One way to remember when to use the semi-colon is that it's similar to what happens when you create an abstract vs. regular methods in an abstract class in Java. For methods that are *not* abstract, you define the method body inside curly braces. But for methods that *are* abstract, you replace the method body with just a semi-colon. For instance:

```
public abstract class MyClass {  
    public abstract void myMethod(); // ** NOTICE!! Ends with a semi-colon  
  
    public void myMethod2() { // this non-abstract method has method body!  
        System.out.println("hi");  
    }  
}
```

The same thing happens here with the child descriptor set: if your descriptor has a child descriptor set, put the set in semi-colons. If your descriptor does *not* have a child descriptor set, you need to show the descriptor has ended by using a semi-colon.

Now that we've learned about all the different parts of a matrix, let's take a look at the actual matrix file next.

## The '.matrix' File

You put your matrix code into a special file with a '.matrix' extension. This file has a regular, Java package statement at the top and should be placed in your java source folder along with all your other java files. As you have seen, .matrix file can have both matrix instructions as well as standard java instructions. And, any java objects you use in your matrices can be imported in using import statements:

```

package com.williespetstore;

import java.util.Date;
import java.text.DateFormat;

MATRIX WilliesPetstore.Content USES (News.Schema) {
    NEWS: {"HomePage News Blurbs"} {
        NEWS.STORY + `Adopt a Pet`: {
            DateFormat.getDateInstance().parse("March 17, 2010"),
            "Adopt a Pet", "lots of text";
            .
            .
            .
        }
    }
}

```

*Example 12 – a WillesPetstore\_\_\$Content.matrix file*

The Matrix and Java instructions are processed by the Hierarchy meta-compiler, which converts the matrix instructions into just pure Java. The meta-compiler will generate a new .java file with the translated code in it. As we previously mentioned, you should think of the Matrix instructions like they were new keywords and operators in the Java programming language. Some of you might have run across compiler pre-processors that did relatively simple text replacement for macro instructions (like C's macros, or the old embedded SQL pre-processors for C). The Hierarchy meta-compiler does much more than simple text macro replacement, and fully understands the Java language. Matrix instructions have been directly added to the Java grammar. This is why you can use full Java expressions in field values. You'll see further usage of the metacompiler aspects of Hierarchy when we look at Hierarchy's embedded files, which require a great mix of Java and Matrix code.

Let's take brief stock of what we've looked at so far. We've learned about matrices and its descriptors. We've also taken apart the descriptor and learned about its individual pieces: the descriptor tag-name, the label, the field set, and the child descriptor set. And lastly, we've learned about the .matrix file. Now, let's move on to how to define the structure of a matrix using schemas.



## Schemas

Let's take another look at the schema for the news blurbs for Willie's Pet Store. Here's what it looked like:

```
package com.williespetstore;

SCHEMA News.Schema {

    DESCRIPTOR +%NEWS {
        FIELD.NAMES: { +%NewsSectionName };
        FIELD.TYPES: { :String };

        DESCRIPTOR +%NEWS.STORY {

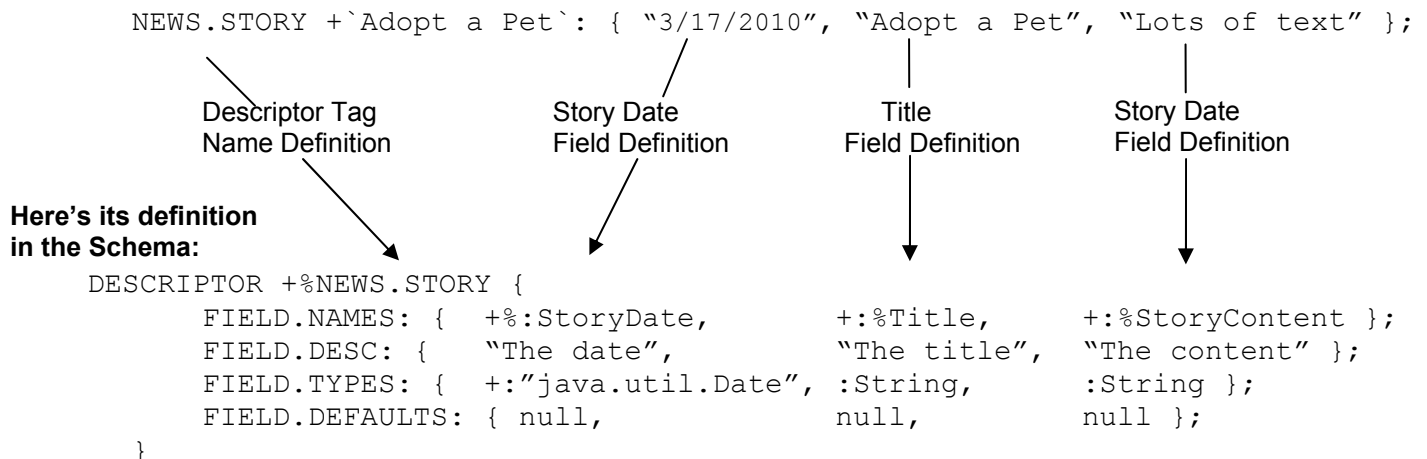
            FIELD.NAMES: { +%StoryDate, +%Title, +%StoryContent };
            FIELD.DESC: { "The date of the story", "The title of the story",
                        "The content of the story" };
            FIELD.TYPES: { +:"java.util.Date", :String, :String };
            FIELD.DEFAULTS: { null, null, null };

        }
    }
}
```

Example 13

In a Schema, to define a descriptor, you create a **schema descriptor**. In the above current schema, examples of schema descriptors would be `DESCRIPTOR +%NEWS` and also `DESCRIPTOR +%NEWS.STORY` (the '+' in front of each of the names means you're creating a new descriptor tag-name. We'll discuss this later in more detail). Let's take a closer look at the schema descriptor for `NEWS.STORY`, and how the different parts of the `NEWS.STORY` descriptor are defined in the schema descriptor. Here's how the different parts map:

In the Matrix, here's a **NEWS.STORY** descriptor:



Example 14

*\*Note, some extra spaces were added to the schema-descriptor definition to make it easier see the relationship of the different fields and their schema settings*

Notice the descriptors, `FIELD.NAMES`, `FIELD.DESC`, `FIELD.TYPES`, and `FIELD.DEFAULTS`. Collectively, we call all these descriptors, the `FIELD.*` descriptors. It may already be evident to you how these `FIELD.*` work, but let's go over them briefly. You define the first field of the `NEWS.STORY`'s fieldset by giving the appropriate schema in the first column of all the different `FIELD.*` descriptor's fieldsets. And, you define the second field value by setting the appropriate schema values in the second column of these fieldsets and so on for each of the fields in your fieldset. This is pretty important to notice, so we'll reiterate this: the column position of the field in the `FIELD.*` fieldsets defines the position of the field when it is used in the actual descriptor's fieldset of a matrix.

Now, what do each of the different `FIELD.*` fieldsets mean? Again, this might already be evident to you, but the way they work are: you define the name of each field in the `FIELD.NAMES` fieldset, and you define the type of each field in the `FIELD.TYPES` fieldset. Then, you can define optional field-descriptions in the `FIELD.DESC` fieldset, and optional default field-values in the `FIELD.DEFAULTS` fieldset.

The next thing to notice is that if you want to define that one descriptor can be a child descriptor of another (so, in our example, `NEWS.STORY` can be a child descriptor of the parent descriptor, `NEWS`), simply put that child's schema-descriptor definition as a child of its parent schema-descriptor. So, in our example, `DESCRIPTOR +%:NEWS.STORY` schema descriptor is a child of the `NEWS` schema descriptor. This means that you'll never see a `NEWS.STORY` descriptor without it being inside a parent `NEWS` descriptor.

The idea here is, by looking at the schema descriptors in the schemas a matrix uses, you should easily be able to see the structure of a matrix's descriptors. To put it another way, it should be very easy for you to see which descriptors are child descriptors of which parents by just looking at the schema. One of the goals the developers of Matrix Programming had when designing the schemas was to make them easy to create and understand. Developers should be able to look at the schema descriptors in a schema and then intuitively be able to see what their corresponding descriptor would look like when used in a matrix.

The last thing to notice is that schemas are matrices as well. Their schema descriptors are just descriptors themselves, and any type of operation you can do on a matrix (such as perform matrix-access operations which are covered in the next section), you can perform on schemas too. This is similar to how xml schemas are also just xml themselves, same idea.

## Defining Field Types

In our schema examples, you may have noticed that to define the type of field, you use an identifier preceded by a `'.'`. For instance, in the `WilliesPetstore.Content` schema, we used `:String` in the `FIELD.TYPES` field set. `:String` is called a **symbol**. Symbols are similar to the symbol objects defined in Ruby (with a few key differences). Briefly, they are programming constants and are used in a very similar way to Java enums. They are used a great deal in Matrix programming. We'll get back to this later.

Note that if you use a primitive type as the type of one of the fields in your schema, use the "symbol with quotes" syntax to specify it – the reason is because primitive types are Java keywords, and can't be used as symbol names directly.

```
FIELD.TYPES:      { :String, :String, :"int"};
```

You may have also noticed that in the `FIELD.NAMES` field set, all the field names were defined with a `'+: %'` right before the field name, for instance, `+:StoryDate` and `+:Title`. In addition, you may have also noticed that we use this same operator when we define the name of a new descriptor, such as in `DESCRIPTOR`

`+%NEWS .STORY`. The `'+: %'` means you're creating and then accessing a new descriptor tag-name. If you remember our discussion on matrices, descriptor tag-names are the name objects used to identify the type of descriptor.

We've been talking a lot about symbols, descriptor tags and labels. In next chapter, we'll go into these three in more detail.

## Chapter 3 - Symbols

### Symbols

In Matrix Programming, Symbols are a new type that have been introduced. They are like Java enums, and are immutable values that Matrix developers use in situations when they need a constant value. To use a symbol, put a ':' before an identifier:

```
:Blue, :Green, :Red, :Black...
```

Actually, before you can use a symbol for the first time, you must create it. To create a new symbol, add a '+' to the symbol:

```
+:Blue, +:Green, +:Red, +:Black...
```

This operator works similar to the 'new' operator in Java. It creates a new symbol object. And, also note that when you create a symbol, you can also access it at the same time too:

```
public void printBlueSymbols() {  
    System.out.println(+:Blue + " is the symbol's color."); // note that you are  
                    // creating and accessing the symbol at the same time!  
  
    // Now that you've created the blue symbol, you can access it without the "+"  
    System.out.println("The symbol is " + :Blue); // notice there is no plus!  
}
```

You use symbols in place of constants or enums. For instance, let's say you're implementing the code for a digitally-controlled oven. So, you might have an Oven class that allows you to set it do different bake modes:

```
public class Oven {  
    public static SYMBOL[] bakeModeTypes = new SYMBOL[] {  
        +:Bake, +:Broil, +:SelfClean };  
  
    SYMBOL bakeMode;  
  
    public void setBakeMode(SYMBOL bakeMode) {  
        this.bakeMode = bakeMode;  
    }  
    public SYMBOL getBakeMode() {  
        return bakeMode;  
    }  
}
```

```

public printBakeMode() {

    if (bakeMode == :Bake)
        System.out.println("Baking...");
    else if (bakeMode == :Broil)
        System.out.println("I'm broiling!");
    else if (bakeMode == :Broil)
        System.out.println("Self-clean.");
}

public static void main(String[] args) {
    Oven myOvenController = new Oven();

    oven.setBakeMode(:Broil);
    oven.printBakeMode();
}
}

```

### Example 17

When you use a symbol or a label for the first time, you need to create it using the "+". But, after it's been created, you can access it directly without the plus. You can include it if you'd like, but this not advisable. By leaving off the plus, you get a limited degree of type safety, because you can only access the symbol if it's already been created. For instance, if you included the plus and accidentally made a spelling mistake, then you won't be accessing the same symbol:

```

+:Blue;
System.out.println(+:blue); // Notice the lower case 'b'. These are two
                           // different symbols!!!

```

When, what you really wanted to do is this:

```

+:Blue;
System.out.println(:Blue); // these are the same symbols. Notice, no plus here

```

But, you can only have one instance of a specific symbol. So, if you try to create a duplicate of an existing symbol, you'll just get the same symbol object. This next example demonstrates this.

```

import net.unconventionalthinking.matrix.symbols.Symbol;

public class MyClass {

    public void testSymbol() {

        Symbol mySymbolVar = +:MySymbol;

        // Here, the '==' test whether these references point to the same object,
        // just like a normal equality test for objects.
        if (mySymbolVar == +:MySymbol); // They do point to the same object! So,
            System.out.println("They're equal!"); // this next statement will be
                                                // executed.
    }
}

```

*This example is done using an “embedded matrix” code file, which we haven’t gone over yet. But, to give you a brief summary of what they are, an embedded matrix code file is a special java file that allows developers to use both java and matrix instructions, and that they typically use to access the matrices and schemas that they write.*

*Example 18*

And so, in this example, what the testSymbol() method does is it creates a symbol and sets it to a Symbol variable. Then, it tries to recreate the same symbol. The important thing to take away here is that since :MySymbol already exists, the second time we call +:MySymbol, it will just return the same symbol object!

Symbols can have multiple parts, separated using the dot operator. For instance, we could create the color symbols with an initial symbol part:

```
Symbol blue = +:Color.Blue;
Symbol red = +:Color.Red;
Symbol brown = +:Color.Brown;

Symbol redishGreen = +:Color.Redish.Green;
```

So all four of these multi-part symbols are symbol objects, and can be used in all the same places a single symbol can.

There are two, main reasons why you’d use multipart symbols over just regular, single symbols. The first is that sometimes it just easier to read a symbol when its multipart:

```
+:TextFormat.FontStyle.Italic  is long and difficult to understand, while
+:TextFormat.FontStyle.Italic  is much easier to read.
```

The second reason is the parts of a symbol allow you to group different child symbols together under a parent symbol. For instance, let’s say you’re creating a word processor application. You can specify the format of a block of text using symbols:

```
+:TextFormat.Color.Black;
+:TextFormat.Color.Blue;
+:TextFormat.Color.Red;

+:TextFormat.Style.Bold;
+:TextFormat.Style.Underlined;
```

And, let’s say you have a method that takes in a text string and a list of formatting symbols, and display a formatted version of the text. You can use the symbol parts to easily identify what type of formatting the symbol belongs to:

```

public void static displayFormattedText(String text, List<Symbol> formatSymbols) {

    for (Symbol formatSym : formatSymbols) {
        // Note: symbol part numbers start at the index 1, not 0.
        if (formatSym.get_PartNumAt(2) == :Style) {
            // This is a style format symbol, do the necessary actions to
            // to properly format the text in this style
            .
            .
            .
        }
    }
}

```

*Example 19*

Multipart symbols are used through out Matrix Programming (in fact, note that a descriptor tag name, `NEWS.STORY`, is a multi-part symbol behind the scenes).

### Quoted Symbols, a Second Way of Defining a Symbol

If you need to define symbols with text that has non-alphanumeric characters in it, then you can put the symbol in quotes. So:

```
+: "This symbol has spaces"
```

If you have a multipart symbol, you can put each part of the symbol in quotes.

```
+: "Hi!". "Ple@se". "don't". "smoke!!"
+: What. "Up?" // Notice! Not all the parts have to be in quotes!
```

Having quotes don't change the value of the symbol. What this means is:

```
+: "mysymbol" == +:mysymbol is a true condition.
```

Now, just to test your knowledge, Here's a question for you:

Look at these three symbol definitions:

```
+: "my.symbol.with.parts"
+: "my". "symbol". "with". "parts"
+: my.symbol.with.parts
```

Are `+: "my.symbol.with.parts"` and `+: "my". "symbol". "with". "parts"` equal?

... The answer is, no, they are not. The symbol, `+: "my.symbol.with.parts"`, is a *single symbol* with no parts, while `+: "my". "symbol". "with". "parts"` is a multi part symbol, with three separate parts!

A follow up question is:

Are `+: "my"."symbol"."with"."parts"` and `+:my.symbol.with.parts` equal?

... The answer is, yes they are. Whether a symbol is defined in quotes or not doesn't change what the symbol's value is. The quotes are only there to let you define symbols with non-alphabetic characters in it.

Did these question make sense to you? I hope they did.

Before we move on to the next section, let's revisit symbol-creation, and symbol type-safety. The type-safety used by symbol creation is not just for a single file, but is application wide. More specifically, if you create a symbol earlier on in meta-compilation process, you can access it without the plus later on. So, for instance, if you create a symbol in a schema, you can access it in your matrix without the plus. So, to show this in an example:

Schema file that uses a symbol:

```
package com.williespetstore;

SCHEMA News.Schema {

    DESCRIPTOR +:%NEWS{
        FIELD.NAMES: { +:NewsWindowBorderColor };
        FIELD.TYPES: { :Symbol };
        FIELD.DEFAULTS: { +:Gray };

        DESCRIPTOR +:%NEWS.STORY {
            .
            .
            .
        }
    }
}
```

*Example 20*

Matrices are metacompiled *after* all the schemas. So, you can use any symbols created in any of your schemas in your matrices *without the plus*:

```
package com.williespetstore;

import SCHEMA com.williespetstore::News.Schema; // we import the schema we're using

MATRIX WilliesPetstore.Content USES (News.Schema) {
    NEWS: { :Gray } { // We don't need to include the '+' here!!!
    }
}
```

*Example 21*



So, we've gone over how symbols work and how to use them. Actually (as we hinted at earlier), there are many other types of symbols besides these standard symbols we just talked about. These other types are child types of the standard symbol that are more specialized and are used in specific situations. Both labels and descriptor tag-names are actually child symbol types, and we'll look at both these two more closely in the next sections.

## Descriptor Tag-Names – a second type of symbol

Descriptor Tag-Names are just a special type of symbol. In fact, in terms of Java objects, symbols are objects that implement the interface, `Symbol_Base` and descriptor tag-names are objects that implement the interface `DescriptorTagName`. And, the `DescriptorTagName` interface is a child of the `Symbol` interface, inheriting directly off of it. What this means is that Descriptor tag-names can work just like symbol objects, but are used in different situations. Before we go into this in more depth, let's figure out how to use descriptor tag-names by taking a look at them in matrices and in schemas.

So, let's take a look at a shortened version the `WilliesPetstore.Content` matrix:

```
MATRIX WilliesPetstore.Content USES (News.Schema) {
  NEWS: {"HomePage News Blurbs"} {
    NEWS.STORY +`Adopt a Pet`: { "3/17/2010", ... }
  }
}
```

All the elements in bold are descriptor tag-names. So, `NEWS`, `NEWS.STORY`, `News.Schema`, `MATRIX`, and `+:%WilliesPetstore.Content` are all descriptor tag-names. Let's focus on just the `NEWS` and `NEWS.STORY` descriptor tag-names for now. As you can see, when you use a descriptor tag-name to access a descriptor, you don't precede the name with any special operator, and just use the name directly. But, if we look back at the schema where these two descriptors get defined:

```
SCHEMA News.Schema {
  DESCRIPTOR +:%NEWS{
    FIELD.NAMES: { +:NewsSectionName };
    FIELD.TYPES: { :String };

    DESCRIPTOR +%NEWS.STORY { ... }
  }
}
```

*Example 22*

You *do* use the `:%` for descriptor tag-names in the schema! Why is this? It's because when we use descriptor tag names as a part of matrix, we know that these are descriptor tag-names because of the syntax – the descriptor tag-names are being used in Matrix creation to define descriptors. But, when we use descriptor tag-names like they're objects (for instance, as a value of a variable, or in this situation, being passed in to the label field of a schema descriptor), then we can't tell by the syntax what they are just by the name, so we need to add the descriptor tag-name operator, `:%`, to mark it as a descriptor tag-name:

```
DescriptorTagName myDescTagName = NEWS; // BAD!! - is NEWS a variable? Or a the
// name of a class? We can't tell just by the syntax.
```



```
DescriptorTagName myDescTagName = +:%NEWS; // GOOD!
```

So what to take away here is that when we use the descriptor tag-name like an object, we need to precede it with a `:%` (or if you're creating a new descriptor tag-name, precede it with a `+:%`). But, when we use them to access different descriptors in a Matrix (or a schema which is, as you may remember, is also just a matrix), then you can just use the simple name with no operator.

We mentioned earlier how descriptor tag-names are just a special type of symbol. Descriptor tag-names behave almost exactly the same way as symbols, but are used in different situations. As we mentioned earlier, the `DescriptorTagName` interfaces that defines descriptor tag-name objects is a child interface of the `Symbol` class. So, any operation you can perform on a symbol, you can perform on a descriptor tag-name. BUT, Descriptor tag-name objects are entirely separate from Symbol objects, and no descriptor tag-name object will ever equal a Symbol object.

```
System.out.println(+:symbol_One == :symbol_One); // true
System.out.println(+:symbol_Two == :symbol_One); // false

System.out.println(+:%descTagName_One == :%descTagName_One); // true
System.out.println(+:%descTagName_Two == :%descTagName_One); // false

System.out.println(+:%sometext == +:sometext); // FALSE!!! - NOTICE we are
// comparing a Descriptor Tag Name with a symbol!!!
```

### *Example 23*

Accessing descriptor tag-names using the `:%` operator is useful in certain situations. One common situation is when you have a descriptor object and you want to see if its descriptor tag-name is equal to some specific descriptor tag-name object:

```
DESCRIPTOR myDescriptorVar = ...; // Don't worry about this var too much.
// It's just a variable for Descriptor objects.

if (myDescriptorVar.getDescriptorTagName() == :%NEWS.STORY) {
    System.out.println("This is a NEWS.STORY descriptor. Printing the story out: ");
    // Print out the news story
    .
    .
    .
}
```

As you can see, descriptor tag names are used very similarly to symbol objects, except they are specialized symbols which are only used in descriptors.

## Labels – A Third Type of Symbol

Like descriptor tag-names, labels are another symbol. Labels represent Java objects that implement the `Label` interface. The `Label` interface extends the parent interface, `Symbol_Base`. Labels are used in the label field of a descriptor:

```

MATRIX WilliesPetstore.Content USES (News.Schema) {

    NEWS: {"HomePage News Blurbs"} {

        NEWS.STORY +`Adopt a Pet`: { DateFormat.getDateInstance().parse("March 17, 2010"),
            "Adopt a Pet", "Your chance to adopt a lonely doggie is right around
            the corner. Come by Willie's Pets on Sunday to find the personality
            that's a perfect match for you!"
        };

        NEWS.STORY +`Fanciful Dog Food`: {
            DateFormat.getDateInstance().parse("March 17, 2010"),
            "20% off Fanciful Dog Spread", "For one day only, come and pick up
            your tub of Fanciful Dog Spread! A truly refined and tasteful
            topping your dog will dream about."
        };
    }
}

```

*Example 24*

All the elements in bold are labels. Notice that the syntax of a label uses back-ticks characters, not single quotes! And, so to define a label, simply surround a block of text using back ticks. Labels work the same as descriptor tag-names and symbols. To create a new label, you need to put a '+' in front of the label. And, just like descriptor tag-names and symbols, no label will ever equal a symbol or descriptor tag-name:

```

System.out.println(+:symbol_One == :symbol_One); // true
System.out.println(+`label_One` == `label_One`); // true
System.out.println(+:%descTagName_One == :%descTagName_One); // true

System.out.println(:one == `one`); // false
System.out.println(`one` == :one); // false

```

*Example 25*

## Summary of How to Use Symbols

At first, when and where to use all the different types of symbols in what situations can be confusing (especially in schemas). Once you get comfortable with them it becomes second nature, but at first, it requires a little extra attention. So let's create a summary of the different types of symbols and when to use them. Refer back to this summary if you have any questions about the symbol types.

### Regular Symbols

**Usage:** You use the regular symbol objects in situations where you'd use a constant:

```
public void printHtmlTable(Symbol tableStyle, String[] content) {
    if (tableStyle == +:Simple) {
        // Print simple table style
        ...
    } else if (tableStyle == +:Fancy) {
        // Print fancy table style
        ...
    }
}
```

Syntax	Examples
:MySymbol or +:MySymbol	:MySymbol, +:My.MultiPart.Symbol
:"MySymbol" or +:"MySymbol"	+: "My \$ymb@1", :"Hi There"."My"."Friend!!"
You can put the text of a symbol in quotes. The reason is so you can put spaces or other characters in the text.	

## Descriptor Tag-Names

**Usage:** Descriptor Tag-Names are used as the tag name of descriptors. There are two types of syntax for the two types of ways you use descriptor tag names:

### 1. Descriptor Tag-Names with *no* Descriptor Tag-Name Operator: For use in Matrix Definition and Matrix Access

When you use descriptor tag-names in matrix definitions or in matrix accesses, you use the descriptor tag names *without* using the descriptor tag-name operator (so, just to make this clear, there is no `:%` preceding the tag name). Here are some examples:

#### Matrix Definition

*The '→' point out all the different descriptor tag names that don't have a descriptor tag name operator.*

```
→ MATRIX WilliesPetstore.Content USES (News.Schema) {  
    → NEWS {  
        → NEWS.STORY + `Adopt a Pet`: { ... }  
    }  
}
```

#### Matrix Access

```
→ WilliesPetstore.Content-→NEWS-→NEWS.STORY:>Title
```

↓                      ↓                      ↓

Notice that in this examples, for all the Descriptor Tag-Names pointed out by the '→', none of these are preceded by the `:%` operator.

### 2. Descriptor Tag-Names *with* Descriptor Tag-Name Operators: For use as Java objects

You use Descriptor Tag-Names with the Descriptor Tag-Name Operator when you use Descriptor Tag-Names like Java objects. For instance, here's a code snippet you might find in an embedded Java file:

```
if (myDescriptorVar.get_DescTagName() == :%NEWS) {  
    System.out.println("Print out the News Section!");  
} else if (myDescriptorVar.get_DescTagName() == :%CONTACT.INFO) {  
    System.out.println("Print out the Contact Info!");  
}
```

The descriptor tag-names, `:%NEWS` and `:%CONTACT.INFO`, are being used like a Java objects. In this situation, you need the `:%` operator.

Syntax	Examples
<code>:%MyDescTagName</code> or <code>+:MyDescTagName</code>	<code>:%NEWS,    +:%NEWS.STORY</code>

## Labels

**Usage:** Like descriptor tag names, there are two usages for labels: For matrix definition / matrix access, and as a Java object. But, unlike Descriptor Tag-Names, there is only one syntax for using them, you must always use `` (back ticks) to create labels. Here's the syntax:

Syntax	Examples
<code>`My Label`</code> or <code>+:`My Label`</code>  * Remember, these are back ticks, <i>not</i> single quotes!	<code>+`10% Discount on Dog Food`</code> , <code>+`News Stories`.`Discounts`.`Dog Food`</code>

Here are the different types of usages of Labels:

### 1. Labels for use in Matrix Definition and Matrix Access

#### Matrix Definition

*The '↓' points out all the different labels*

```
MATRIX WilliesPetstore.Content USES (News.Schema) {  
    NEWS {  
        NEWS.STORY +`Adopt a Pet`: { ... }  
    }  
}
```

Here, we've defined a `NEWS.STORY` descriptor with an `+`Adopt a Pet`` label.

#### Matrix Access

```
WilliesPetstore.Content->NEWS->`10% Discount on Dog Food`:>Title
```

Here, we're using a label to do access a matrix.

### 2. Labels for use as Java objects

You can use Labels like Java objects. For instance, here's a code snippet you might find in an embedded Java file:

```
DESCRIPTOR myNewsStoryDesc = getNewsStoryDescriptorFromMatrix(MyMatrix);  
if (myNewsStoryDesc.get_Label() == `Discounted Dog Food Story`) {  
    System.out.println("Print out Discounted Dog Food News Story!");  
} else if (myNewsStoryDesc.get_Label() == `Heroic Dog Rescues Owner`) {  
    System.out.println("Print out the Heroic Dog News Story!");  
}
```

The labels, ``Discounted Dog Food Story`` and ``Heroic Dog Rescues Owner``, are being used like a Java objects.

## Chapter 4 - Schemas

We talked about Schemas in Chapter 2, in the tutorial. In this chapter, we'll take a deeper look at them. Note that we're going to repeat a lot of the information we already talked about in Chapter 2 (some of it is repeated word for word). But, make sure you keep reading because we expand each topic in more detail.

### Defining Descriptors

So, let's get back to the Willie's Pet Store example. Here's what the schema looked like:

```
package com.williespetstore;

SCHEMA News.Schema {

    DESCRIPTOR +%NEWS {
        FIELD.NAMES: { +%NewsSectionName };
        FIELD.TYPES: { :String };

        DESCRIPTOR +%NEWS.STORY {

            FIELD.NAMES: { +%StoryDate, +%Title, +%StoryContent };
            FIELD.DESC: { "The date of the story", "The title of the story",
                "The content of the story" };
            FIELD.TYPES: { +:"java.util.Date", :String, :String };
            FIELD.DEFAULTS: { null, null, null };

        }
    }
}
```

*Example 13*

In a Schema, to define a descriptor, you create a **schema descriptor**. In the above current schema, examples of schema descriptors would be `DESCRIPTOR +%NEWS` and also `DESCRIPTOR +%NEWS.STORY`. As we now know the '+' in front of each of the names means you're creating a new descriptor tag-name. Let's take a closer look at the schema descriptor for `NEWS.STORY`, and how the different parts of the `NEWS.STORY` descriptor are defined in the schema descriptor. Here's how the different parts map:

In the Matrix, here's a **NEWS.STORY** descriptor:

```
NEWS.STORY +`Adopt a Pet`: { "3/17/2010", "Adopt a Pet", "Lots of text" };
```

Descriptor Tag  
Name Definition

Story Date  
Field Definition

Title  
Field Definition

Story Date  
Field Definition

Here's its definition  
in the Schema:

```
DESCRIPTOR +%NEWS.STORY {  
  FIELD.NAMES: { +%:StoryDate,      +%:Title,      +%:StoryContent };  
  FIELD.DESC: {   "The date",        "The title",   "The content" };  
  FIELD.TYPES: {  +:"java.util.Date", :String,      :String };  
  FIELD.DEFAULTS: { null,            null,         null };  
}
```

#### Example 14

*\*Note, some extra spaces were added to the schema-descriptor definition to make it easier see the relationship of the different fields and their schema settings*

Notice the descriptors, `FIELD.NAMES`, `FIELD.DESC`, `FIELD.TYPES`, and `FIELD.DEFAULTS`. Collectively, we call all these descriptors, the `FIELD.*` descriptors. You define the first field of the `NEWS.STORY`'s fieldset by giving the appropriate schema in the first column of all the different `FIELD.*` descriptor's fieldsets. And, you define the second field value by setting the appropriate schema values in the second column of these fieldsets and so on for each of the fields in you fieldset. This is pretty important to notice, so we'll reiterate this: the column position of the field in the `FIELD.*` fieldsets defines the position of the field when it used in the actual descriptor's fieldset of a matrix.

Now, what do each of the different `FIELD.*` fieldsets mean? For each field, you define the name of the field in the `FIELD.NAMES` fieldset, and you define its type in the `FIELD.TYPES` fieldset. Then, you can define optional field-descriptions in the `FIELD.DESC` fieldset, and optional default field-values in the `FIELD.DEFAULTS` fieldset.

The next thing to notice is that if you want to define that one descriptor can be a child descriptor of another (so, in our example, `NEWS.STORY` can be a child descriptor of the parent descriptor, `NEWS`), simply put that child's schema-descriptor definition as a child of its parent schema-descriptor. So, in our example, `DESCRIPTOR +%:NEWS.STORY` schema descriptor is a child of the `NEWS` schema descriptor. This means that you'll never see a `NEWS.STORY` descriptor without it being inside a parent `NEWS` descriptor.

The idea here is, by looking at the schema descriptors in the schemas a matrix uses, you should easily be able to see the structure of a matrix's descriptors. To put it another way, it should be very easy for you to see which descriptors are child descriptors of which parents by just looking at the schema. One of the goals the developers of Matrix Programming had when designing the schemas was to make them easy to create and understand. Developers should be able to look at the schema descriptors in a schema and then intuitively be able to see what their corresponding descriptor would look like when used in a matrix.

The last thing to notice is that schemas are matrices as well. Their schema descriptors are just descriptors themselves, and any type of operation you can do on a matrix (such as perform matrix-access operations which are covered in the next section), you can perform on schemas too. This is similar to how xml schemas are also just xml themselves, same idea.



## Defining Field Types

In our schema examples, you may have noticed that to define the type of field, you use an identifier preceded by a `:`. For instance, in the `WilliesPetstore.Content` schema, we used `:String` in the `FIELD.TYPES` field set. `:String` is called a **symbol**. Symbols are similar to the symbol objects defined in Ruby (with a few key differences). Briefly, they are programming constants and are used in a very similar way to Java enums. They are used a great deal in Matrix programming. We'll get back to this later.

Note that if you use a primitive type as the type of one of the fields in your schema, use the “symbol with quotes” syntax to specify it – the reason is because primitive types are Java keywords, and can't be used as symbol names directly.

```
FIELD.TYPES:      { :String, :String, :int};
```

In the `FIELD.NAMES` field set, all the field names were defined with a `+:%` right before the field name – for instance, `+:StoryDate` and `+:Title`. In addition, we use this same operator when we define the name of a new descriptor, such as in `DESCRIPTOR +%NEWS.STORY`. We've already discussed this in the previous chapter, but as a reminder, the `+:%` means you're creating and then accessing a new descriptor tag-name. Also, if you want to use an object type like `java.util.Date`, you can do this by creating a new symbol object with the object's full class name.

```
FIELD.TYPES:      { +:"java.util.Date", :String, :int};
```

The metacompiler expects this type name in a single set of quotes. ***It's a pretty common error to pass in object types wrong!*** Here's a clarification of what does and doesn't work as a symbol for an object type definition:

<code>+: "java.util.Date"</code>	😊	Good object type name definition.
<code>+: "java"."util"."Date"</code>	😞	Bad – And note, this is <i>not</i> equal to the previous one!
<code>+: java.util.Date</code>	😞	Bad – This also does not equal the first one!

Oh, and just in case your wondering, the reason you don't need a plus for the primitive types like `:int` is because Hierarchy pre-creates commonly used symbols, like those for the primitive types.

Now, here's a question side question for you:

Why is `+: "java.util.Date"` not equal to `+: "java"."util"."Date"`?

The answer is that `+: "java.util.Date"` is a *single symbol* with no parts, while `+: "java"."util"."Date"` is a multi part symbol, with three separate parts!

A follow up question is:

Are `+: "java"."util"."Date"` and `+: java.util.Date` equal?

The answer is, yes they are. Whether a symbol is defined in quotes or not doesn't change what the symbol's value is. The quotes are only there to let you define symbols with non-alphabetic characters in it.

If you still have questions, refer back to the previous chapter on symbols.

Also, you can also use classes that you've defined in your own project or in jars. It works exactly the same:

```
FIELD.TYPES:      { +:"com.myproj.MyType", : "String", : "int" };
```

Now that we've talked about how to define a schema, let's discuss how to use it. If you want to use a schema in a matrix, you need to make it available to the matrix. This works the same way as how, in Java, a class is made available to another class, using an import statement. Let's take a look.

## Using a Schema in a Matrix: Importing the Schema

The code for a schema definition is placed in a '.schema' file. And, if a matrix wants to use a schema, just like how for a regular Java class, a matrix must import the schema in. In addition though, the matrix must then refer to this imported schema in its `USES` clause. So, if we look at our sample matrix again:

```
package com.williespetstore;

import java.util.Date;
import java.text.DateFormat;

import SCHEMA com.williespetstore::News.Schema

MATRIX WilliesPetstore.Content USES (News.Schema) {
    .
    .
    .
}
```

*Example 15 - WilliesPetstore\_\_\$Content.matrix*

In this .matrix file, we added an `import SCHEMA` instruction. This import statement works the same as Java imports. After you made the import, you can now use the schema by its simple name, `News.Schema`, throughout the rest of the file. And also like Java imports, if the schema is in the same package as the matrix, you don't need to import it. You can use it right away.

And, just to send this point home, remember though, after you import the schema in, you still need to add it to the matrix `USES` clause!

For the name of the schema, you specify the full schema name, separating the simple schema-name from its package using the scope-resolution operator `::` operator (similar to the one you may have seen in C++).

Package name	Simple Schema Name
↓	↓
com.williespetstore::News.Schema	

But, also similar to Java classes, you don't have to import a schema before you use it, you can specify it using the full schema name. Let's look at this next.

## Using a Schema in a Matrix: by Specifying the Full Schema Name

Notice that in the previous example, in its import statement for the schema, you specify the full schema name, separating the simple schema-name from its package using the `::` operator. But, similar to Java, you don't have to import a schema to use it; you can simply specify the schema in using its full schema name. For instance, in the above example, we could have the matrix use the schema through its full schema name:

```

package com.williespetstore;

import java.util.Date;
import java.text.DateFormat;

// NOTICE! Not importing the schema. Instead, use the fully qualified schema name.
MATRIX WilliesPetstore.Content USES (com.williespetstore::News.Schema) {
    .
    .
    .
}

```

*Example 16*

It's pretty straightforward. Anytime you use a schema (or a matrix), you can specify it by its full name.

In this last section of this chapter, we're going to summarize how to use the different symbol types in a schema. When you create a schema for the first time, it can be a little confusing to know in what situations to use each of the different types of symbols. Let's go over this now.

## Summary of Usages of the Different Types of Symbols in a Schema

In a schema, at first, it can be a little confusing to know what type of symbol to use in a certain situation. This section will provide a summary of symbol usage in schemas (you may want to bookmark this page and refer back to it when you first start creating schemas of your own).

The way we'll do this simply by show a sample schema and pointing out the different types of symbols used.

```
package com.williespetstore;
```

For the name of the Schema, use a name with **no operators** preceding it.

```
SCHEMA News.Schema {
```

For the name of a newly defined descriptor use the **Descriptor Tag-Name operator, :%** – In this situation, we're using a Descriptor Tag-Name like a Java object. It's being passed into the DESCRIPTOR definition's parameter. Since it's being used like a Java object, precede it with the descriptor tag-name operator, :%

```
DESCRIPTOR +%NEWS.STORY {
```

For defining the field names, again, we're using a Descriptor Tag-Name like a Java object. Precede the name with the **descriptor tag-name operator, :%**

```
FIELD.NAMES: { +%StoryDate, +%Title, +%StoryContent };
FIELD.DESC: { "The date of the story", "The title of the story",
              "The content of the story" };
```

For defining the types of fields, pass in a regular symbol object. But, use the **:"" symbol operator by wrapping the type name in quotes!** Why? For primitive types like `int`, if you don't wrap it in quotes (for ex. `:int`), you'll get a syntax error because `int` is a reserved word in Java (actually, here, `String` is the one primitive type that doesn't need to be wrapped in quotes, because `String` is not a reserved word. But, just to be consistent, it's a good idea)...

```
FIELD.TYPES: { +:"java.util.Date", :String, :String };
```

... And, for object types like `java.util.Date`, the metacompiler expects the type name in a single set of quotes. **It's a pretty common error to pass in object types wrong!** So:

<code>+: "java.util.Date"</code>	😊 Good
<code>+: "java"."util"."Date"</code>	😞 Bad – And note, this is <i>not</i> equal to the previous one!
<code>+: java.util.Date</code>	😞 Bad – This also does not equal the first one!

```
FIELD.DEFAULTS: { null, null, null };
```

```
    }
}
```

Refer back to this summary if you ever have problems with the symbol types in a schema.

## **Let's sum up what we've gone over so far**

So, we've learned how matrices and their descriptors work. Specifically, we've learned about the different parts of a descriptor: the descriptor tag-name, the label, the field set, and the child-descriptor set. We also learned how to define descriptors using schemas. And, lastly, we learned about the different types of symbols: standard symbols, descriptor tag-names, and labels.

These last three chapters dealt with how to create matrices and schemas. The next couple of chapters will deal with more details specifically regarding matrices, and then also on how to access matrices in your Java code.

## Chapter 5 - Matrices

This is a short chapter. In the tutorial in Chapter 2, we already had a good discussion about the key concepts of matrices, and for most of your questions on matrices, refer back to that earlier chapter. But, we have some extra topics we'd like to discuss. Let's get started.

### Matrices, along with Schemas are a new data-type for Java

I know we've mentioned this a couple of times before, but just to make sure we start our discussion of matrices right, we'll repeat what we said in the tour we took in the first chapter:

"Conceptually, you should think of a Matrix as a new data type for Java. It's on the same level as a regular class. In fact, matrices aren't static like XML documents, but, instead like instances of a Java class, are mutable. You can dynamically add, remove, and modify values at runtime. You can even add whole new subtrees. We created Hierarchy with the idea of building on what's already in Java, and matrices were designed to naturally fit in as a new, third data-type. In between primitive types (like int's) and classes, matrices are a third, new data-type available for you to use.

And briefly, to finish this conceptual picture of matrices, behind the scenes, a matrix is just a collection of regular, Java objects. This collection has Descriptor objects, which have FieldSet objects, and to wrap the whole thing is a Matrix object."

Very important to know this, as it'll give you a better understanding of how matrices fit in at a higher level to the Java language!

### The ITEM Descriptor

There is a descriptor type that you can use at any time in your matrices, it's called the ITEM descriptor. This descriptor is special. You use it by creating a descriptor that doesn't have a descriptor tag-name and just has a label:

```
package com.williespetstore;

MATRIX WebForm.Registration USES (Web.Form, Database) {

    `First Name` { // look! no descriptor-tag name, just a label!
        FORM.INPUT.TEXTBOX { 30 };
    }
    `Last Name` {
        FORM.INPUT.TEXTBOX { 30 };
    }
    `Address 1`{
        FORM.INPUT.TEXTBOX { 60 };
    }
    :
    :
}
```

The labels in bold are the ITEM descriptors. And, just to drive this point home, the metacompiler knows that they are ITEM descriptors because they only have a label defined. But, you can also specify the ITEM descriptor tag name if you'd like:

```

package com.williespetstore;

MATRIX WebForm.Registration USES (Web.Form, Database) {

    ITEM `First Name` { // this descriptor is equivalent to the above example!
        FORM.INPUT.TEXTBOX { 30 };
    }
    ITEM `Last Name` {
        FORM.INPUT.TEXTBOX { 30 };
    }
    ITEM `Address 1`{
        FORM.INPUT.TEXTBOX { 60 };
    }
    .
    .
}

```

These last two examples were equivalent, and you can create an ITEM descriptor with or without the ITEM descriptor-tag name. Now, why would you want to use the ITEM descriptor? It's really just a convenience thing. There are situations where you want to group a bunch of descriptors together. The ITEM descriptor is very good for this. A real world usage of this is the one we were starting to show in the previous example, creating a registration page for a web form. Each field in the form has a lot of different kinds of information associated with it. Let's create a more in depth version of the matrix from the previous example:

```

package com.williespetstore;

MATRIX WebForm.Registration USES (Web.Form, Database) {

    `First Name`
    FORM.REQUIRED: { +:IsRequired };
    FORM.INPUT.TEXTBOX { 30 };
    // Table, column name, type
    DB.COLUMN: { +:Customer, +:First_Name, :String }

    `Last Name`
    FORM.REQUIRED: { +:IsRequired };
    FORM.INPUT.TEXTBOX { 30 };
    DB.COLUMN: { +:Customer, +:Last_Name, :String }

    `Gender`
    FORM.REQUIRED: { +:IsRequired };
    FORM.CONTROL.RADIOBOX { "GenderRadioBox" } {
        RADIO.ITEM { "Male", "Male", :Checked };
        RADIO.ITEM { "Female", "Female" };
    }
    DB.COLUMN: { :Customer, +:Gender, :int }

    `Address 1`
    FORM.REQUIRED: { +:IsRequired };
    FORM.INPUT.TEXTBOX { 60 };
    DB.COLUMN: { +:Customer, +:Address1, :String }
    .
    .
    .
}
}

```

We can use this matrix in a servlet or JSP page to generate a multi-page form for a web page fairly easily. Matrices seem to naturally work well for this situation. In fact, we expand on this same, web form example in a later chapter. In this later chapter, we also show the code for the JSP that uses this matrix. To read this longer discussion of this type of usage, go to chapter 10 and take a look at “Case #3 – Universal Data Definition.” (but, you may want to wait until you’ve read the next chapter first, the discussion will make more sense then).

In the next chapter, we discuss how to access a matrix in your Java code using embedded Java files.



## Chapter 6 - Accessing Matrices in your Embedded Java Code

### Main Concepts of Embedded Java Files

Let's now answer the question, how do you access a matrix from your Java code? After a developer has created his matrices and schemas in his .matrix and .schema files, he accesses his matrices by creating special java files that allow matrix instructions to be embedded in the Java code. These special Java files have a ".mjava" extension. ".mjava" files are meta-compiled by the Hierarchy metacompiler after all the .schema and the .matrix files. We call these .mjava files **embedded Java files**.

As we mentioned in a previous section, when we say "embedded", we don't mean a file where a simple text search and replace of matrix instructions is done, like "C" style macros, but it's as if the matrix instructions are an extension to the Java language itself. As we've have seen through out this tutorial, the Hierarchy metacompiler fully understands the Java grammar and has been extended with instructions and operators for accessing matrices. To show you what how you can fully use Java in embedded files, let's take a look at an example of an embedded Java file. Continuing with the news blurbs example we worked on earlier, let's create a version of it that prints out the page to standard out. You can use matrices with jsp's and servlets, but to keep things simple, we'll do a console version of the news home page that uses `System.out`. First though, let's refresh our memory of the full, Willie's Pet Store matrix:

```
package com.williespetstore;

import java.util.Date;
import java.text.DateFormat;

MATRIX WilliesPetstore.Content USES (News.Schema) {

    NEWS: {"HomePage News Blurbs"} {

        NEWS.STORY +`Adopt a Pet`: {
            DateFormat.getDateInstance().parse("March 17, 2010"), "Adopt a Pet",
            "Your chance to adopt a lonely doggie is right around the " +
            "corner. Come by Willie's Pets on Sunday to find " +
            "the personality that's a perfect match for you!"
        };

        NEWS.STORY +`Fanciful Dog Food`: {
            DateFormat.getDateInstance().parse("March 10, 2010"),
            "20% off Fanciful Dog Spread",
            "For one day only, come and pick up your tub of Fanciful Dog " +
            "Spread! A truly refined and tasteful topping your dog will " +
            "dream about."
        };
    }
}
```

*Example 26*

To access this matrix, you first need to create an embedded Java file to your project. This embedded Java file will use the ".mjava" file extension. The way you should think about these .mjava files are they are just regular Java files that also allow you to use our extended syntax to access and manipulate matrices.

You use the matrix access operator, '->' to access a matrix. Let's access the `Adopt a Pet` News Story descriptor:

```
package com.mypackage;

import MATRIX com.williespetstore::WilliesPetstore.Content;

public class AccessTest {

    public void accessMatrix() {
        // notice the last access is using just the label
        DESCRIPTOR descVar1 = WilliesPetstore.Content->NEWS->`Adopt a Pet`;

        // Alternately, you can access descriptors with both the descriptor tag-name
        // and the label:
        DESCRIPTOR descVar2 = WilliesPetstore.Content->NEWS->NEWS.STORY `Adopt a Pet`;

        // This prints `true`
        System.out.println("descVar is the same object as descVar2: " +
            (descVar==descVar2));
    }
}
```

*AccessTest.mjava*  
*Example 27*

*\* For now, we'll ignore the DESCRIPTOR descVar's in this example and get back to them later.*

As you can see, a .mjava file is really just an extended Java file. More specifically, in your .mjava files, you can create classes with methods and variables just as you would with a regular Java file, but, you can also use our extended syntax to access matrices. And, as you can see in this example, you access descriptors by their descriptor tag-name using the -> operator.

*Each time you use the descriptor-access operator, ->, it should resolve to a single descriptor!* But, what happens when you use the operator and there are multiple descriptor with the same descriptor tag-name to pick from? For instance, `WilliesPetstore.Content->NEWS->NEWS.STORY`, resolves to two descriptors. Then, in this situation, you use a label to uniquely choose one. So, for our example, we picked between the two `NEWS.STORY` descriptors in the `NEWS` descriptor using the label, ``Adopt a Pet``.

More over, you cannot even pick more than one descriptor using the descriptor-access operator, '->' (well, you can, but there is additional syntax you'll need that we haven't gone over. We'll get back to this later). If you create an access expression that resolves at any point to multiple descriptors, it is an error:

```
DESCRIPTOR descVar3 = WilliesPetstore.Content->NEWS->NEWS.STORY; // ERROR!!
// resolves to 2 descriptors
```



This error may be caught at either meta-compilation time or at run time. The reason is because a matrix can be created using three creation types: as a static matrix, a dynamic matrix, or a statically-defined, dynamically run matrix. A static matrix doesn't change, meaning descriptors can not be added, modified or removed from it, so

the meta-compiler can determine if a matrix has an improper access during metacompilation time. So, static matrices, meta-compiler will throw an error if it detects a single-descriptor matrix-access that resolves to more than one descriptor. But, for dynamic matrices that can be modified at runtime, the metacompiler cannot determine at metacompilation time if an access is improper, so will not detect an improper access until runtime, when the access is actually tried.

But, the default for a matrix is a “statically-defined, dynamically run matrix”. What this means that during the metacompilation process, a matrix is considered static, and all *static* matrix accesses are checked for improper accesses, throwing errors for those it finds (when we say *static* matrix access, what we mean is that a matrix access defined at metacompile time. You can also do dynamic matrix access, we’ll take about this later). But, a statically-defined, dynamically-run matrix can also be modified at runtime. The program can add, modify and remove descriptors and fields as needed.

Now that we’ve figured out how to select a descriptor, let’s access the field values out of a descriptor’s field set. This is done with the field access operator, ‘:>’.

```
String title = WilliesPetstore.Content->NEWS->`Adopt a Pet`:>Title;
```

And, if we wanted to print out this info, here’s what the code would look like:

```
System.out.println(WilliesPetstore.Content->NEWS->`Adopt a Pet`:>Title);
System.out.println("_____");
System.out.println(
    (WilliesPetstore.Content->NEWS->`Adopt a Pet`:>StoryDate).toString()
);
System.out.println("\n" +
    WilliesPetstore.Content->NEWS->`Adopt a Pet`:>StoryContent);
```

#### Example 28

Notice how when we access the `StoryDate` field, since this returns a java `Date` object, we are able to call the `Date` object’s `toString()` method off of the field access. As you can see though, we have to wrap this access in parenthesis. This is necessary because descriptor tag-names (and all other types of symbols) can have a ‘.’ in between different parts of the name. For instance, in our schema, we could define the `StoryContent` field as `Story.Content`. These name parts with their dots make it very difficult for the metacompiler to determine when we’re doing object-variable access off of a matrix-field access. So, the rule is that when you want to access a variable or a method off of a matrix access, surround the matrix access first with parenthesis.

Now, it’d be nice if we didn’t have to re-type the full access (`WilliesPetstore.Content->NEWS-> ...`) with each access we do. One way to do this in certain situations is by using descriptor variables.

## Descriptor Variables

Descriptor variables let you save a reference to a specific descriptor. You may have noticed these being used before in the examples. For instance, in example 27 on the previous page, the variables: `descVar1`, `descVar2`, and `descVar3` were all descriptor variables. These are **non-access-typed** descriptor variables, meaning they are references that can point to a descriptor in any part of a matrix. But, you can also create

descriptor variables that can only hold descriptors with a given access path is through a specific matrix. For instance:

```
DESCRIPTOR<WilliesPetstore.Content->NEWS->NEWS.STORY> adoptAPetDescVar =  
    WilliesPetstore.Content->NEWS->`Adopt a Pet`;
```

We put the access path through the matrix in the descriptor variable itself. Notice that in the descriptor variable's access-type, we didn't need to have each part of the access resolve to a specific descriptor, we just needed to enter in the general path. So, in this example, instead of specifying the label, `Adopt a Pet` in our last descriptor access, we used the more general `NEWS.STORY`, which, if you remember, resolves to two descriptors! The reason we can do this is because we aren't do a real access in the access-type, we're just letting the metacompiler know what the general path through the matrix is. What's happening here is that the metacompiler needs to know what possible descriptors in a matrix you could set this descriptor variable to. The reason is because that when you try to access a descriptor or field off of a descriptor variable, the metacompiler needs to know what the next access options are. The metacompiler uses the access-type value to determine these next access-options.

So, syntactically when you create an access type for a descriptor, a way to remember how to create them is: In each path element, you don't need to use any labels to identify specific descriptors, just the descriptor tag-names.

Using a matrix-access type in a descriptor variable may seem like a lot of redundant code, typing most of the access string both in the access-type of the variable and in the access itself, but if you reuse this variable over and over again, it reduces the amount of code you'll have to write. Let's rewrite our previous example to use a descriptor variable.

```
DESCRIPTOR<WilliesPetstore.Content->NEWS->NEWS.STORY> adoptAPetDescVar =  
    WilliesPetstore.Content->NEWS->`Adopt a Pet`;  
  
System.out.println(adoptAPetDescVar:>Title);  
System.out.println("_____");  
System.out.println( (adoptAPetDescVar:>StoryDate).toString() + "\n");  
System.out.println(adoptAPetDescVar:>StoryContent);
```

#### *Example 29*

Not only is there less code, but this should also be a performance improvement over the previous example. The descriptor variable holds a reference to the `Adopt a Pet` descriptor, and doesn't have to redo the access to this descriptor with each field access as in the previous version of this code.

## **Multi-Access, Part 1**

For our news story Java console application, we probably don't want to print out each news story one by one, hard coding the access to each descriptor we need. It'd be nice if we could select a set of descriptors and then loop over this set, printing each one. In Matrix Programming, to select a set of descriptors (or fields), we use the matrix-access operator with multi-access. In our running example, the code to select all the news story descriptors in the parent news descriptor looks like this:

```
MatrixSet<Descriptor> newsStoryDescriptors =  
    WilliesPetstore.Content->NEWS->NEWS.STORY{*};
```

The opening and closing braces, {}, after the `NEWS.STORY` descriptor tag-name means we're accessing multiple descriptors, and by using a \*, we're saying we're accessing all the child `NEWS.STORY` descriptors in

the parent NEWS descriptor. If you want to filter some of the child descriptors out narrowing down the set, you can add filtering code inside the braces of the multi-access operator (we'll return to this later). The result of this expression will return in a special set called a MatrixSet which has many of the same behaviors as a Set **class** found in the com.java.util package. It holds all of the descriptors that the multi-access resolves to.

You can also have multiple levels of multi-access, and include fields as well:

```
MatrixSet newsStoryDateFields =
    WilliesPetstore.Content->NEWS->NEWS.STORY{*}>StoryDate{*};
```

But, once you start doing multi-access, you can not return to single access. This results in a metacompilation error:

```
Date newsStoryDate =
    WilliesPetstore.Content->NEWS->NEWS.STORY{*}>StoryDate; // ERROR!!
    // Didn't continue to do multi access!
```

Now, in our new story application, we can grab all the news stories and loop over each one. In this next code example, we'll combine the use of descriptor variables with multi-access:

```
for (DESCRIPTOR<WilliesPetstore.Content->NEWS->NEWS.STORY> newsStoryDesc :
    WilliesPetstore.Content->NEWS->NEWS.STORY{*}) {

    System.out.println(newsStoryDesc:>Title);
    System.out.println("_____");
    System.out.println( (newsStoryDesc:>StoryDate).toString() + "\n");
    System.out.println(newsStoryDesc:>StoryContent + "\n\n");
}
```

### *Example 30*

This for loop is just a standard, Java for-each loop. The result of the `WilliesPetstore.Content->NEWS->NEWS.STORY{*}` multi-access is a MatrixSet, which can be iterated over and fed into the last parameter of the for-each. The loop iterates over this set of descriptors, assigning each descriptor object to the `newsStoryDesc` variable.

We didn't mention this earlier, we thought it'd better to leave this till later. But before you can use matrix, you need to make sure you can access. Let's talk about this now:

## Using a Matrix in an Embedded Java File: Importing the Matrix

We saw that in the previous chapter on schemas, if a matrix needs to use a schema, the matrix can import the schema in. Embedded files work the same way. If an embedded file needs to use a matrix, you simply need to import the matrix in. The only difference is instead of using the, "import SCHEMA" instruction, you use the, "import MATRIX", instruction. Let's jump back to the embedded-code example for this chapter that was introduced a few pages ago. If you take another look, you'll see it looks like this:

```
package com.mypackage;

import MATRIX com.williespetstore::WilliesPetstore.Content;

public class AccessTest {
```

This import MATRIX instruction works the same as Java imports, you can now use this matrix by its simple name, `WilliesPetstore.Content`, throughout the rest of the file. And also like Java imports, if the matrix is in the same package as the embedded file, you don't need to import it. You can use it use it right away.

And, just like with using a schema in a matrix, you don't need to import a matrix, you can use it by specifying the full matrix name. Let's talk about this next.

## Using a Matrix in an Embedded Java File: by Specifying the Full Matrix-Name

Similar to Java, you don't have to import a matrix to use it. You can simply specify the full matrix name. For instance, in the above example, we could use the matrix through its full matrix-name with out importing it first:

```
package com.mypackage;

public class AccessTest {

    public void accessMatrix() {

        DESCRIPTOR descVar1 =
            com.williespetstore::WilliesPetstore.Content->NEWS->`Adopt a Pet`;
        .
        .
        .
    }
}
```

Notice, we're accessing the matrix using the full matrix name, and there is no import of the `WilliesPetstore.Content` matrix before we use it!

Now, what happens if some type of error occurs during a matrix access? For instance, since we are using "statically-created, dynamically-run" matrices, we are allowed to modify the matrices at runtime. Descriptors are just Java objects, and have methods that let you add and remove descriptors at runtime. We could create a situation where a matrix access might fail:

```
// Remove the `Adopt a Pet` child descriptor.
(WilliesPetstore.Content->NEWS).remove_ChildDescriptor(executeInfo, `Adopt a Pet`);

// Now, try to access the `Adopt a Pet`. During metacompilation, The metacompiler
// will not know that this access will fail since there is no
// `Adopt a Pet` descriptor anymore! This error will only be detected at
// runtime.
System.out.println(WilliesPetstore.Content->NEWS->`Adopt a Pet`->Title); // ERROR!
```

### Example 31

What will happen in this situation? How are access errors detected and processed? Normally, in Java, this is done using exceptions. In Matrix Programming, this is done using matrix annotations. Matrix annotations are not to be confused with the annotations we use on our Java class and method declarations. These are entirely different, and next we'll go over matrix annotations along with multi-access filters (which are defined similarly to Matrix annotations).

## Annotations

When you access a matrix, you can specify an error handler using matrix annotations. You use annotations in matrix-access during situations that you'd normally think would throw some type of exception. For instance, in the previous example, this code tries to access the removed ``Adopt a Pet`` descriptor, which you normally would think would result in a null pointer exception. In matrices, the error is passed to a **Matrix annotation handler** which is defined at the end of a method declaration. So, for example:

```
public PetStoreAccessor {

    public String getAdoptAPetTitle() {

        // Remove the `News` child descriptor from the matrix.
        // NOTE: the executeInfo object is automatically added in as a static
        // variable to each class in an .mjava file
        (WilliesPetstore.Content).remove_ChildDescriptor(executeInfo, :%NEWS);

        // Now, try to access the `NEWS`.
        // ERROR! Will access a missing descriptor!
        return WilliesPetstore.Content-<*1>NEWS->`Adopt a Pet`:>Title);

        ANNOTATIONS {
            *1 {
                return null;
            }
            DEFAULT: {
                return null;
            }
        }
    }
}
```

*Example 32*

There are three main parts to an annotation: the annotations block, the annotations handlers (which is inside the annotations block), and the annotation reference. To make this even clearer, let's take our annotation example code and label it with the three different parts:

```

public String getAdoptAPetTitle() {

    // NOTE: the executeInfo object is automatically added in as a static
    // variable to each class in an .mjava file
    (WilliesPetstore.Content).remove_ChildDescriptor(executeInfo, :%NEWS);

                                This is the annotation reference
                                ↓
    return WilliesPetstore.Content-<*>NEWS->`Adopt a Pet`:>Title);

    ANNOTATIONS { ← This is the annotation block
        *1 { ← This is the *1 annotation handler
            return null;
        }
        DEFAULT: { ← This is the DEFAULT annotation handler
            return null;
        }
    }
}

```

Example 33

As you can see, you define your annotations in the `ANNOTATIONS` block, defined at the end of a method, **and even after the return statement**. Here, we've defined an annotation handler with the reference name of `'*1'`, and we refer to that annotation reference in our matrix access using a "matrix-access with annotation reference" operator :

```
MyDescriptor-<*>MyReference name>AnotherDesc
```

Generally what happens is if an error occurs while trying to do the descriptor access at the `'*1'` reference, this error along with its reference is passed to the annotations block, which then hands it off to the matching annotations handler. Then, the annotation handler processes the error, and can (amongst other things) stop the access and return a value for the entire access expression. To understand the process more fully, let's go over the steps that occur during the matrix-access error in Example 32:

1. The program tries to access the NEWS descriptor off of the `WilliesPetstore.Content` matrix.
2. There is no NEWS descriptor, so the program will pass the error along with the annotation reference `'*1'` to the annotations block.
3. The annotations block searches through its top level annotations handlers for a matching reference, which it finds with the `'*1'` annotations handler.
4. The program executes the code in the `'*1'` annotation handler. A return statement sends control back to the matrix access, along with the value in the return.
5. The matrix access is stopped, and the returned value becomes the new value of the matrix access expression (so in our example, the value returned of our failed access will be `null`).

But, you don't have to stop the current access. If you use the `ACCESSOR.RETURN` statement instead of standard return, the value returned will then be the value of the *current access*, and the statement will try to continue.

```

public String getAdoptAPetTitle() {

    // Remove the `News` child descriptor from the matrix.
    // NOTE: the executeInfo object is automatically added in as a static
    // variable to each class in an .mjava file
    (WilliesPetstore.Content).remove_ChildDescriptor(executeInfo, :%NEWS);
}

```



```

// Now, try to access the `NEWS`.
// ERROR! Will access a missing descriptor!
// But, the annotation returns a different NEWS descriptor and
// tries to continue the access
return WilliesPetstore.Content-<*1>NEWS->`Adopt a Pet`:>Title);

ANNOTATIONS {
    *1 {
        // returns a different NEWS descriptor
        ACCESSOR.RETURN MyOtherNewsMatrix->NEWS;
    }

    DEFAULT: {
        return null;
    }
}
}

```

#### Example 34

Before we go on, we should explain that even though we've defined our annotation reference name as `*1`, with a number, but you don't have to use a numbers, you can use an identifier as well. We could have used something like `*One` or `*Num_One` as the name of our annotation. In fact, an annotation reference is just a symbol object, the same symbol objects that we've used in the matrices and schemas seen throughout these examples.

So, to return to processing access errors, what would happen if an error occurred later in the access? For instance, at the ``Adopt a Pet`` part of the access?

```

public String getAdoptAPetTitle() {

    // Remove the `Adopt a Pet` child descriptor.
    // NOTE: the executeInfo object is automatically added in as a static
    // variable to each class in an .mjava file
    (WilliesPetstore.Content->NEWS).remove_ChildDescriptor(executeInfo,
                                                            `Adopt a Pet`);

    // Now, try to access the `Adopt a Pet`.
    // ERROR! Will access a missing descriptor later in the access
    return WilliesPetstore.Content-<*1>NEWS->`Adopt a Pet`:>Title);
}

```

#### Example 35

During the matrix access, each access after the first access (with its *explicit* reference) is given a hidden, *implicit* reference. If we “un-hid” the implicit references, it would look something like this:

Original access:

```
WilliesPetstore.Content-<*1>NEWS->`Adopt a Pet`:>Title
```

Redoing the access with “un-hidden” implicit references:

```
WilliesPetstore.Content-<*1>NEWS-<*1.1>`Adopt a Pet`:<*1.2>Title
```

- A base name which is the explicit reference. So, for our example, the base name is `*1`

- An access counter that is added to the base name. Something like \*1.1

In the previous sample code with its `Adopt a Pet` access error, let's add annotation handlers for the implicit references:

```
public PetStoreAccessor {

    public String getAdoptAPetTitle() {

        // NOTE: the executeInfo object is automatically added in as a static
        // variable to each class in an .mjava file
        (WilliesPetstore.Content->NEWS).remove_ChildDescriptor(executeInfo,
                                                                `Adopt a Pet`);

        // Now, try to access the `Adopt a Pet`.
        // ERROR! Will access a missing descriptor later in the access
        return WilliesPetstore.Content-<*1>NEWS->`Adopt a Pet`:>Title);

    }

    ANNOTATIONS {
        *1 {
            *1.0 { // This handler is for the NEWS descriptor access
                return null;
            }
            *1.1 { // This handler is for the `Adopt a Pet` descriptor access
                // We probably don't want to do this in real code, which is
                // return some other descriptor, but we do it here just for the
                // sake of this tutorial
                ACCESSOR.RETURN WilliesPetstore.Content->NEWS->`Fanciful Dog Food`;
            }
            *1.2 { // This handler is for the Title field access
                return "my default title";
            }
        }

        DEFAULT: {
            return null;
        }
    }
}
```

Example 36

What will happen here is when this method is called, the matrix access:

WilliesPetstore.Content-<\*1>NEWS->`Adopt a Pet`:>Title will result in an error at `Adopt a Pet`, which will create the implicit reference, `\*1.1`. The error along with the implicit reference will be passed to the annotation block. The annotation block will search its handlers for a matching reference, finding the \*1.1 handler, which will be executed. This handler returns the `Fanciful Dog Food` descriptor back to the matrix access, which then will be used to continue the access. And so, the final result of the access is that the title of the `Fanciful Dog Food` descriptor will be returned.

What happens if we have just a \*1 handler defined with none of the implicit handlers? The annotation block first tries to find an exact match for the reference. If it doesn't find one, it tries just the base name. So, in this case, the base handler would process the error (then, if it doesn't find a base handler, it falls to the default handler). This next example shows the base handler being used:

```

public String getAdoptAPetTitle() {

    // NOTE: the executeInfo object is automatically added in as a static
    // variable to each class in an .mjava file
    (WilliesPetstore.Content->NEWS).remove_ChildDescriptor(executeInfo,
                                                            `Adopt a Pet`);

    // Now, try to access the `Adopt a Pet`.
    // ERROR! Will access a missing descriptor later in the access
    return WilliesPetstore.Content-< *1 > NEWS-> `Adopt a Pet`:>Title);

    ANNOTATIONS {
        *1 { // the `Adopt a Pet` error would fall back to this base handler
            return null;
        }
        DEFAULT: {
            return null;
        }
    }
}

```

*Example 37*

For this code, the \*1 handler would be executed, returning null for the entire access expression.

You can also define a default handler that catches any errors that aren't caught by any of the other handlers. It's recommended that you always define one.

```

public String getAdoptAPetTitle() {

    // NOTE: the executeInfo object is automatically added in as a static
    // variable to each class in an .mjava file
    (WilliesPetstore.Content->NEWS).remove_ChildDescriptor(executeInfo,
                                                            `Adopt a Pet`);

    // Now, try to access the `Adopt a Pet`.
    // ERROR! Will access a missing descriptor later in the access
    return WilliesPetstore.Content-< *1 > NEWS-> `Adopt a Pet`:>Title);

    ANNOTATIONS {
        DEFAULT: { // the `Adopt a Pet` error would fall to being processed here
            return null;
        }
    }
}

```

*Example 38*

## Multi-Access, Part 2 – Filters

As you may remember, a multi-access statement can define filters that can narrow down the set of items that are returned during a multi-access. Here's a multi-access statement with out a filter:

```
MatrixSet<Descriptor> newsStoryDescriptors =
    WilliesPetstore.Content->NEWS->NEWS.STORY{*};
```

To define a filter, you would do this similarly to how you would create an annotation. You create a reference to the filter in the `{}` operator, and then define the filter in the `FILTERS` section of the `ANNOTATION` block:

```
public PetStoreAccessor {

    public MatrixSet<Descriptor> selectSomeNewsStories() {

        return WilliesPetstore.Content->NEWS->NEWS.STORY{*MySimpleFilter};

    }

    ANNOTATIONS {
        DEFAULT: {
            return null;
        }

        FILTERS {
            // This simple filter creates a set that only has the
            // the first news story in it.
            *MySimpleFilter {
                MatrixSet<Descriptor> newsStory_Full =
                    (MatrixSet<Descriptor>)curr_ValueSet_Full;
                MatrixSet<Descriptor> newsStory_Filtered = new
                    MatrixSet<Descriptor>();

                newsStory_Filtered.add(newsStory_Full.get(0));

                // this is a hidden variable you should set when
                // your filter is executed.
                enteredFilter = true;
                return newsStory_Filtered;
            }
        }
    }
}
```

Example 39

The `*MySimpleFilter` is called when the `NEWS.STORY` descriptor set is created. In the filter, you grab the current set from the `curr_ValueSet_Full` hidden variable. You then can create a new set of filtered descriptors, returning this set back to the multi-access statement.

## Matrix access with a dynamically generated label

A pretty common task that you might need to do is when you access a matrix, you may want to dynamically pass in the label used during the access at runtime. This is possible using “dynamic label access”. Here’s an example. Let’s look at a matrix that describes a line of products for a company:

```
package com.mychemicalcompany.website;

MATRIX Website.Content {

    PRODUCT.INFO +`UltraChem Toothpaste`: {
        // product description text, department name
        "UltraChem Toothpaste",
        "Oral Hygiene"
    }

    PRODUCT.INFO +`NeonBrite Facial Scrub`: {
        // product description text, department name
        "NeonBrite Facial Scrub will make your skin glow! Try it today!",
        "Facial Products"
    }
    :
    :
    :
}

Website$__.$Content.matrix
```

Now, let’s create a method that selects a specific `PRODUCT.INFO` descriptor using a string for the product name. This string is converted into a label using the symbol factory object:

```
public class MyEmbeddedClass {

    public static String getProductDesc(String productName) {
        // Dynamically create a new label using the product name string
        Label productNameLabel = appControl
            .symbolControl.singleSymbol_Factory
            .createNewLabel (productName, false);

        // pick PRODUCT.INFO descriptor using a DYNAMIC label that’s passed in
        return Website.Content->PRODUCT.INFO [productNameLabel]:>Description;

        ANNOTATION {
            DEFAULT {
                return null;
            }
        }
    }
}
```

It’s pretty straightforward. Notice that the accessing of `PRODUCT.INFO` has a set of brackets where you can pass in a variable that holds a label object: `PRODUCT.INFO [productNameLabel]`. This is called “matrix

access using a dynamic label.” And, note that in terms of performance, this type of access should be very fast. In terms of worst case access, it should be constant time –  $O(1)$ . The reason is because for each descriptor, all its child descriptors are indexed by label using a hash.

## Modifying Fields and Descriptors in Matrices

It’s easy to get into an XML mindset when using matrices, thinking of them as pretty much as static data, that doesn’t change at runtime. But, matrices are not static and are actually very easy to change.

In an .mjava file, to set a field to a new value, you simply do a field access and then an assignment:

```
public class ChangeMatrix {  
  
    public void changeMatrixValue() {  
        WilliesPetstore.Content->NEWS->NEWS.STORY->`Adopt a Pet`:>Title =  
            "Go and Adopt a Lonely Pet Today!";  
    }  
}
```

This simple class accesses the `WilliesPetstore.Content` matrix and makes changes to the `title` field of the ``Adopt a Pet`` descriptor to a new value. Matrix access fully understands the assignment operator.

You can also add and remove descriptors from a matrix. There are a couple of ways to do this. The easiest way is probably to do a matrix access to access the *parent* descriptor of the descriptor you’d like to remove. *Then, wrap this entire access in parenthesis!* After you’ve wrapped the access in parenthesis, you can then call the `remove_ChildDescriptor()` method on the parent descriptor object:

```
↓                ↓  
(WilliesPetstore.Content->NEWS).remove_ChildDescriptor(executeInfo,  
                                                         `Adopt a Pet`);
```

You need to wrap the access in parenthesis because you’re going to call one of the methods on the descriptor object. And, without the parenthesis, the syntax of the method call can get confused with the matrix access syntax (specifically, the `.`` that starts the method call is what causes the problem).

Notice that in the `remove_ChildDescriptor()` method call, you’re passing in an object, `executeInfo`, and the label, ``Adopt a Pet``. This label is used to pick the child descriptor you want to remove. The object, `executeInfo`, is an object that is automatically included in every .mjava file. In your .mjava files, you can access this object at anytime, and is used by your application to determine what happened during any matrix-related operations.

We’ve discussed how to remove descriptors, but as we mentioned, you can also add descriptors to matrices too. Before you can add a descriptor though, you need to create the descriptor object that you’ll be adding. We won’t discuss how to do this at this point, as this syntax for creating descriptors is being worked out right now, and will be released in a future version of Hierarchy. But we just wanted to make you aware that this is something that will be supported in the future...

... Actually though, even without this matrix operators for creating descriptors, you can still create descriptors programmatically, using the class libraries in the `Hierarchy.jar`. It’s pretty complex, and something we’ll discuss in future versions of this document. If it’s a feature you need to use now, please contact the developers of Hierarchy for an explanation.

So, you may be asking yourself when would I want to assign values to, or change descriptors in a matrix? The answer should be fairly often. You should try to think of matrices as just another data structure in Java, one that is specialized for holding large, hierarchical data sets. It shouldn't be thought of as static XML that doesn't change often, but used in situations where you'd often used to use a Java class or an array.

## **Limitations on working with embedded Java files**

Currently, the Hierarchy metacompiler is a beta product and certain features in Java are not currently supported. Here's a list of the current limitations of the metacompiler:

- In an mjava file, you can not have any inner classes.
- Also, you can't have any extra, non-public classes either (following your main class definition).
- You can't put the .mjava file in the default package. It must have a package!

These features will be supported in the future. But for now, we hope you can work around them. And, consult the FAQ for more information. It contains a full list of the limitations and problems you might encounter while working with .mjava files.

## **You've now learned all the main concepts!**

At this point, you've learned all the main concepts of Matrix Programming: you've learned both how to create matrices by creating schema and matrix files, and then how to access them using the matrix-access operator, descriptor variables, annotations and multi-access filters. We've gone over a lot of material, and (if we've done our job) you should feel comfortable with the syntax for using matrices in your Java code. Now, that last major piece is to learn the practical information on how to work with the metacompiler, so you can properly meta-compile, debug, and run your Matrix applications.

# Chapter 7 – Using the Metacompiler: How to metacompile and debug your code

## The Matrix Metacompilation Process

We've already talked about the different parts of the metacompilation process throughout this tutorial (such as schemas are metacompiled before matrix files, which are metacompiled before embedded files), but now, we are going to go over all the different stages of the process in more detail. The reason this information is important is because when you work with matrices, in order to properly metacompile and then especially to debug problems in your matrix code, it's important to know what is built when and where any output files are generated. This is a very long chapter, but an important one. Good luck!

To start with, let's go over all the phases in the metacompilation process at a high-level:

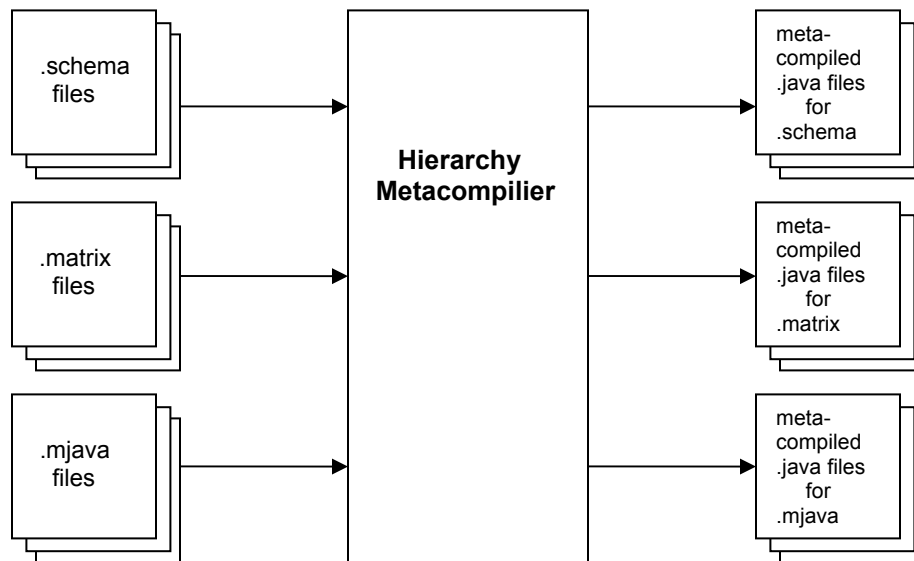
### The Matrix Metacompilation Process

1. **Metacompile all schemas** – Grab all the .schema files in your project and run them through the Hierarchy metacompiler. The metacompiler translates the code into .java files. And then, the metacompiler invokes javac to compile them generated Java into .class files.
2. **Generate all “Specialized FieldSets”** - In your schemas, you use descriptor definitions to define descriptors and their fieldsets. For all these fieldsets definitions, the metacompiler generates the specialized .java code, creating a class for each fieldset definition (Again, after the Java files are generated, javac is invoked to compile them into .class files).
3. **Metacompile all matrices** – Grab all the .matrix files in your project and run them through the metacompiler, generating the translated .java files (ditto).
4. **In your Embedded Java Files: Collect type information** – Grab all the .mjava files and search through them, gathering all the type info for all the variables you use. Note that this grabs type info on *all* types variables, both Java object types and Matrix Programming types like the DESCRIPTOR<> variable. This info is used for processing the Matrix Programming types.
5. **Metacompile all Embedded Java files** – Grab all the .mjava files again and run these through the metacompiler, generating .java files.

At this point, you'd think the metacompiler would always do the final compilation of the generated Java files for embedded files into .class files. But, for embedded files, this final compilation is *optional!* (Sometimes, you might want to let an Ant script do this final compilation, or the IDE).

We've discussed metacompilation before, so this should sound familiar, but just to make sure we understand the key concepts, during each of the metacompilation steps (so steps 1, 2, 3, & 5), we're taking “meta code” files, such as a .schema or a .matrix file, and sending them through the metacompiler, creating .java equivalents. First, let's revisit the diagram we introduced way back in chapter two that shows the inputs and outputs of the metacompiler.





As you can see, the metacompiler takes in matrix file types and generates Java equivalents. You then use the generated Java files directly in your own Java projects.

### Location of the Outputted .java and .class Files

The metacompiler has two build directories that it outputs in results to:

- One directory for generated `.java` files that come out of the metacompilation of the metacode files (`.matrix`, `.schema`, `.mjava`).
- Another directory for the final `.class` files that are created when the generated Java files are compiled by `javac`.

You tell the metacompiler where these directories are by setting them up in the “`hierarchy.properties`” file. You can find further information on these settings in the section on “**Setting up the Metacompiler**” found later in this guide.

### The Debugging Process

Debugging an error in your code using the metacompiler is a bit more complex than debugging a regular Java compiler error. For most errors caught by the metacompiler, you simply need to read the error message to determine what’s wrong. For syntax errors found in your metacode (`.matrix`, `.schema`, or `.mjava` files), the metacompiler will often tell you exactly the line number in your metacode file and the type of syntax error. For instance, in your matrix file, if you forget to close a descriptor’s field set, the metacompiler will output a syntax error:

```
package mypackage;

MATRIX MyMatrix USES (Test.Schema) {
    TEST.DESRIPT { "value 1", "value 2" ;
}
```

Forgot to include closing brace! 😞

This is the error that would be output by the metacompiler:

```
*** ERROR ***

In the file: 'MyMatrix.matrix', in the package: 'mypackage',
had an error trying to parse the file, on line 4, near the token, ';', at
position 42:
    [4,42] expecting: '}'
```

Here, the metacompiler tells you exactly where and what the problem is. When the metacompiler does catch an error in the metacode file (.matrix, .schema, .mjava), it's relatively straightforward to fix these types of bugs. *But, some errors are not detected until later in the metacompilation process, only when the generated Java files are compiled into .class files.* In this situation, when the error is in the generated Java file, it isn't actually the metacompiler that catches the error, it's `javac`. This is because even though the metacompiler seems like it's doing this final compilation for you, all that really happens is it is simply invoking `javac` on the generated Java files and outputting the results to you.

In the future, we hope that the metacompiler will catch all the errors you make during the metacompilation of the metacode files, but for now, you'll occasionally have to open a generated Java file to track down an error. Let's say you accidentally use the wrong class name for a variable in an embedded file:

```
package mypackage;

public class EmbeddedCodeTest {
    BadClass var; // The class, BadClass, doesn't exist!
}
```

The error that the metacompiler would output is:

```
Compiling Generated Java-Files into Class files:
Output is:
mypackage\EmbeddedCodeTest.java:5: cannot find symbol
symbol   : class BadClass
location : class mypackage.EmbeddedCodeTest
    BadClass var;
    ^
1 error
```

Notice that this error isn't for one of the metacode file types, it's for one of the generated Java files, `EmbeddedCodeTest.java`. This means if you want to track down the error, you need to open the generated Java file, `EmbeddedCodeTest.java`, *not* its associated Embedded code file, `EmbeddedCodeTest.mjava`.

And since this error is in the Java file, this also means that the error wasn't caught by the Hierarchy metacompiler, but by `javac`!

What's happened is when the metacompiler metacompiled the `.mjava` file, `EmbeddedCodeTest.mjava`, into its generated Java file, it didn't catch the bad class name error! The reason it misses this error is because the metacompiler tends to only catch errors in the Matrix Programming instructions, not the standard Java instructions. Generally, it lets the Java compiler, `javac`, catch these.

Next, the metacompiler calls `javac` to do the final compilation of the generated Java files into `.class` files. Now, during this second phase of compilation, `javac` has found the bad class name error that wasn't caught by the metacompiler.

The generated Java code is fairly readable, and after learning a few more concepts, which we'll talk about shortly, you should find it relatively easy to understand and navigate through the generated, Java code, allowing you to fairly quickly solve bugs found during this stage of metacompilation. In fact, we've found that once you've gotten pretty good with the metacompiler, you'll find you can solve almost any matrix-related error pretty quickly, whether it's found during meta-compilation or final Java compilation.

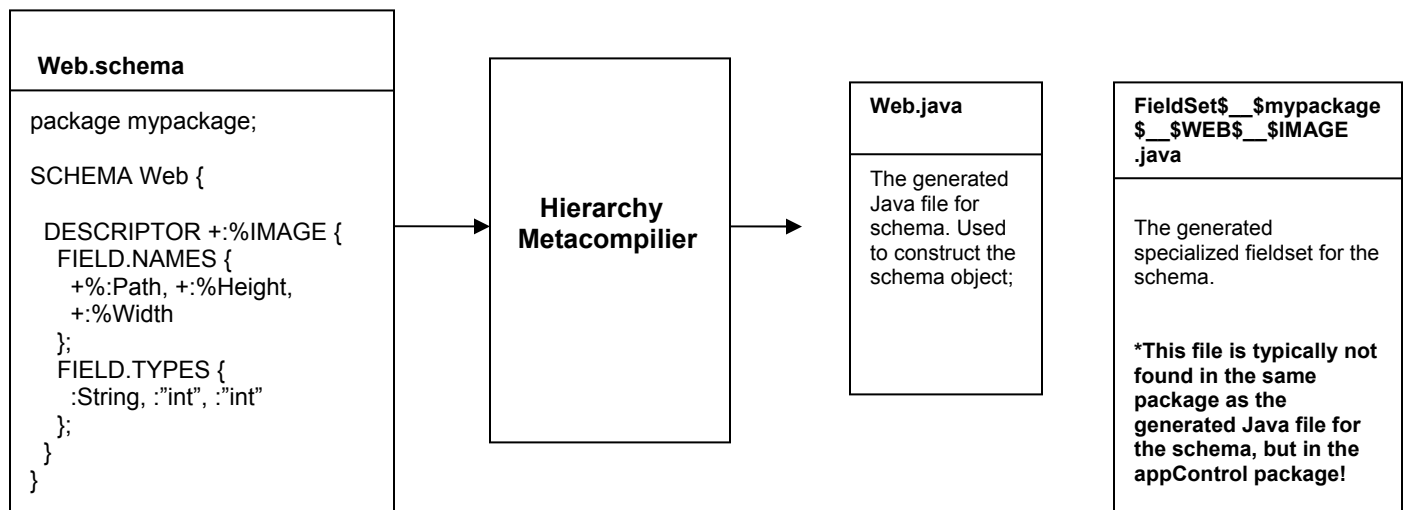
Now that we've got a good understanding of what goes in and out of the metacompiler, let's dive a little deeper and take a look at the generated matrices and schemas.

## Internal Structure of Matrices and Schemas

If you look at the generated Java code for matrices and schemas, what you'll probably realize is that matrices and schemas are *not* some special byte-code that are generated by the metacompiler, but are simply regular Java objects. They are built using classes found in the `Hierarchy.jar`, specifically, in the `net.unconventional.matrix` package.

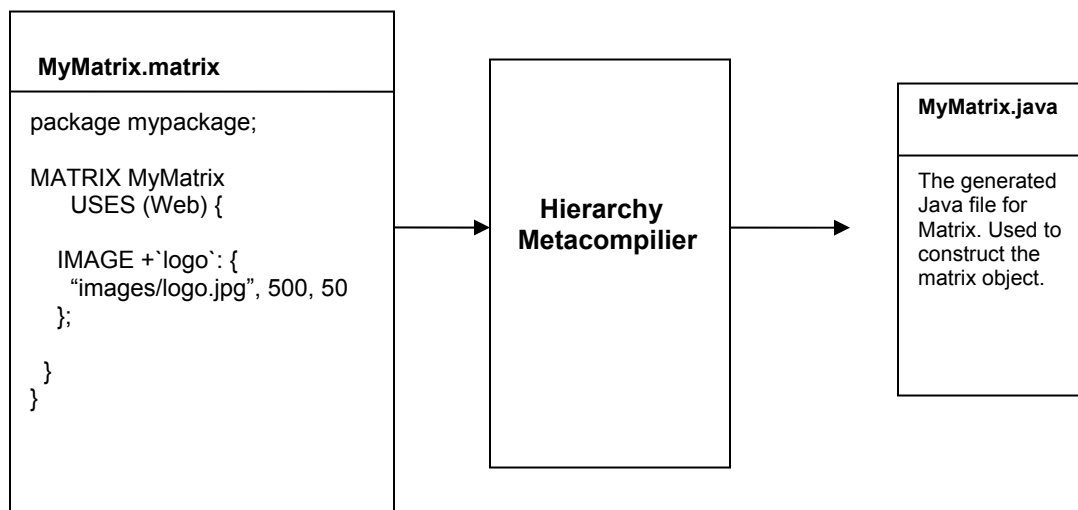
But, before we actually look inside a generated Java file, let's go through each of the metacode file types and see what happens at a high-level when they get metacompiled. First, let's look at the `.schema` and the `.matrix` file types. Here's what goes in and out of the metacompiler for each type.

First, here are the inputs and outputs for the schema:



*Schema Metacompilation Input and Outputs*

Next, here are the inputs and outputs for a matrix:



*Matrix Metacompilation Input and Outputs*

The reason we're talking about schemas and matrices together is because schemas are just a specialized form of matrix, and having a good understanding of the build process and structure of a matrix gives you a good understanding of the build process and structure of a schema. You should be able to see this intuitively by simply comparing a schema to a matrix. If you open a .schema file and compare it to a .matrix file, you should notice that the syntax for a schema is exactly the same as that of a matrix, except it starts with the `SCHEMA` keyword instead of the `MATRIX` keyword.

Now, let's take a look at what's generated for a matrix. We can see from the input and output diagram that one file is created: `MyMatrix.java`. When you start a Java application that uses `MyMatrix`, this class is used to construct the actual matrix object. If you open up a generated Java file for a matrix (you can find these in the

sample projects), what you'll see is a class definition with one method defined, `construct()`. This class looks like this:

```
package mypackage;

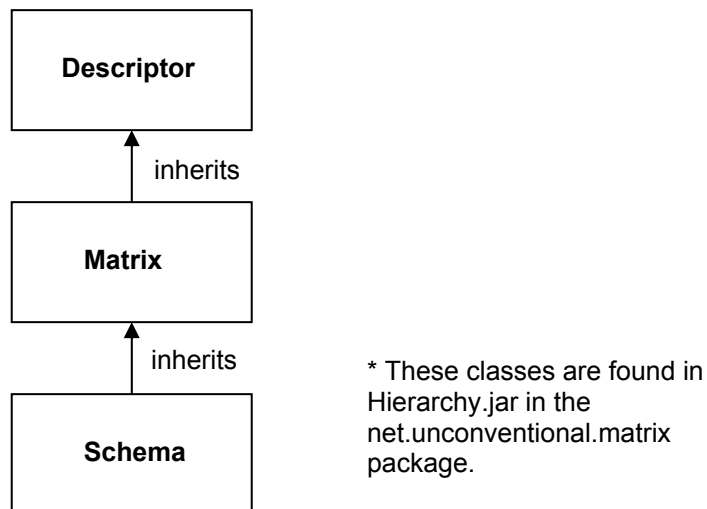
import net.unconventionalthinking.exceptions.*;
import net.unconventionalthinking.matrix.*;
import java.util.Arrays;
import net.unconventionalthinking.lang.*;
import net.unconventionalthinking.matrix.symbols.*;
import java.util.LinkedList;

public class MyMatriximplements MatrixContainer {

    public static Matrix matrix;

    public static Matrix construct(AppControl_Base appControl,
        ExecutionInfo executeInfo) throws Exception_MatrixRuntime_Checked {
        .
        .
        .
    }
}
```

The `construct()` method is called once, during the startup of the application and uses the classes in the `Hierarchy.jar` to build the matrix object. Specifically, it creates one `Matrix` object that contains many nested levels of `Descriptor` objects. You can think of a `Matrix` object as a specialized `Descriptor` object. In fact, the `Matrix` class inherits most of its functionality from the `Descriptor` class. Here's there class diagram, along with the `Schema` class:



This class diagram shows that schema objects are specialized matrix objects, and matrix objects are specialized descriptor object. This makes sense, because a matrix is really just the top level descriptor object, and just like a descriptor, it has a set of child descriptors.

```
MATRIX MyMatrix USES (Web) {
```

↖

**NOTICE!** The syntax structure of the MATRIX definition is the same as the syntax for a descriptor.

↘

```
IMAGE +`logo`: {  
    "images/logo.jpg", 500, 50  
};
```

```
}
```

Now that you've got a good high-level understanding of the structure of a matrix, let's return to the generated Java file for a matrix and look inside the actual `construct()` method. Here's the previous `MyMatrix.java` file we just showed you, but with the `construct()` method filled in. It looks confusing, but it's actually fairly straightforward. We'll explain what to do if an error is found by the Java compiler at certain points in this file.

## The Generated Java for Matrices and Schemas

```
public class MyMatrix implements MatrixContainer {

    public static Matrix matrix;

    public static Matrix construct(AppControl_Base appControl, ExecutionInfo
        executeInfo) throws Exception_MatrixRuntime_Checked {

        LinkedList<MatrixSet<SchemaInfo_Schema>> schemaSet_ScopeStack = new
            LinkedList<MatrixSet<SchemaInfo_Schema>>();
```

First, always look for the comments. They will help guide you through the generated code. They occur when a new descriptor (or matrix) object is created. Here, we start creating the matrix object named 'MyMatrix'.

```
// Creating Matrix with name, MyMatrix _____
```

```
try {
```

This next line sets up the "scope stack" for the schemas that are used by your matrix. If an error occurs here, it means that something is wrong with the USES statement of your matrix. You may not have imported your schema correctly. Refer back to the schemas section for more info.

```
    schemaSet_ScopeStack.add(new MatrixSet<SchemaInfo_Schema>().add(...);
} catch (Exception e) {
    throw new Exception_MatrixRuntime_Checked("while trying to create a new Matrix Or Schema
        descriptor with the name, MyMatrix.", e);
}
```

```
MatrixSet<SchemaInfo_Schema> descriptor68$_X_$MATRIX$_X_$__UsesSchemaSet;
Descriptor descriptor68$_X_$MATRIX;
```

```
try {
```

This line creates a set to hold the schemas that are available to just this newly created descriptor (which, in this case, is the root-level, matrix descriptor). REMEMBER! A matrix object is just a specialized descriptor.

```
descriptor68$_X_$MATRIX$_X_$__UsesSchemaSet = new
    MatrixSet<SchemaInfo_Schema>().add(appControl.schemaControl.schemaIndex_Find(AppSymbols_Sc
        hemaName.mypackage$_CC_$Web));
```

\*\*\* **This is the important line!** \*\*\* It is here that the new matrix object is created and assigned to a temporary descriptor variable. Most errors that do occur in this file happen at these points, when the descriptor objects are created and their field sets are initialized. Notice that for the creation of the matrix descriptor, two important values are passed in:

- the name of the matrix as a label - here, `AppSymbols_Label.MyMatrix`,
- and the package that the matrix belongs in - `AppSymbols_DescTagName.mypackage`

```
descriptor68$ _X_ $MATRIX = (Descriptor)appControl.matrixControl.
    matrixFactory.createNew_Matrix(executeInfo, AppSymbols_Label.MyMatrix,
    AppSymbols_DescTagName.mypackage, true, descriptor68$ _X_ $MATRIX$ _X_ $ __UsesSchemaSet, null,
    true);

    schemaSet_ScopeStack.add(descriptor68$ _X_ $MATRIX$ _X_ $ __UsesSchemaSet);
} catch(Exception e) {
    throw new Exception_MatrixRuntime_Checked("Error creating new descriptor", e);
}
```

Now, we'll start creating the next child descriptor, IMAGE, for this matrix.

```
// Creating Descriptor with descriptor tag, mypackage::Web::IMAGE
MatrixSet<SchemaInfo_Schema> descriptor69$ _X_ $IMAGE$ _X_ $ __UsesSchemaSet;
Descriptor descriptor69$ _X_ $IMAGE;

try {
    descriptor69$ _X_ $IMAGE$ _X_ $ __UsesSchemaSet = new MatrixSet<SchemaInfo_Schema>();
```

\*\*\* **Again, this is another important line** \*\*\* We're creating a new descriptor object, this time for the IMAGE descriptor, using the `createNew_Descriptor()` method. In this method call, we've highlighted a couple of important parameters that are passed in.

- The overall purpose of the code in the 1<sup>st</sup> highlighted text is to access the `Web` schema, and get the descriptor definition for the `IMAGE` descriptor. This is done by:
  1. Calling `schemaIndex_Find()`, to get the `Web` schema object from the schema index.
  2. Then, on the returned schema object, calling `.getChild_SchemaInfoDescriptor()` which grabs the descriptor definition object for `IMAGE` from the schema.
  3. It then passes this descriptor definition object into `createNew_Descriptor()` method. Inside this method, when the new `IMAGE` descriptor is created, the new descriptor is given a reference to this descriptor definition object, using it to determine info about the descriptor (such as how many fields it can have and what the types of the fields are...).
- In the 2<sup>nd</sup> highlighted text, we giving this child descriptor its parent-descriptor, which in this case is the rootlevel, `MATRIX` descriptor-object.

```
descriptor69$ _X_ $IMAGE =
    (Descriptor)appControl.matrixControl.matrixFactory.createNew_Descriptor(executeInfo,
    appControl.schemaControl.schemaIndex_Find(AppSymbols_SchemaName.mypackage$ _CC_ $Web)
    .getChild_SchemaInfoDescriptor(executeInfo,
    AppSymbols_DescTag.mypackage$ _CC_ $Web$ _CC_ $IMAGE),
    null, descriptor69$ _X_ $IMAGE$ _X_ $ __UsesSchemaSet, descriptor68$ _X_ $MATRIX);
```



**\*\* Now, we get the fieldset object from the IMAGE descriptor object. We'll setup this fieldset object with its initial values \*\***

```
mypackage.FieldSetTuple__mypackage$__CC$__Web$__S$__IMAGE descriptor69$__X$__IMAGE$__X$__FieldSet =  
    (mypackage.FieldSetTuple__mypackage$__CC$__Web$__S$__IMAGE)descriptor69$__X$__IMAGE.get_FieldSet  
    _Tuple(executeInfo);
```

And here, we actually do the calls to set the initial values. **If an error occurs here, it's typically because one of the types of the values passed in doesn't match the type for the field!**

For instance, the initial value passed in maybe a String, but the field only accepts primitive int's. This bug would show up here!

And note, **this error is fairly common!!**

```
descriptor69$__X$__IMAGE$__X$__FieldSet.set_Path(executeInfo, "images/logo.jpg");  
descriptor69$__X$__IMAGE$__X$__FieldSet.set_Height(executeInfo, 50);  
descriptor69$__X$__IMAGE$__X$__FieldSet.set_Width(executeInfo, 500);
```

```
schemaSet_ScopeStack.add(descriptor69$__X$__IMAGE$__X$__UsesSchemaSet);
```

Here, we add the IMAGE descriptor to its parent descriptor (which is the matrix object).

```
descriptor68$__X$__MATRIX.add_ChildDescriptor(executeInfo, descriptor69$__X$__IMAGE);
```

```
} catch(Exception e) {  
    throw new Exception_MatrixRuntime_Checked("Tried to create a new descriptor with the name,  
        maintests.samples::WEB.FORM::FORM.REQUIRED, and with an empty label, but had an error", e);  
}  
}
```

And remember, since a schema is a matrix, the generated Java file for a schema is basically the same as that of a matrix (except schemas use method calls that are schema specific instead just for a matrix). Now that you understand the generated Java for a matrix, you should also be able to understand and debug problems that are caught in the generated Java for your schemas now too.

In the next section, we'll take a look at the generated Java for the last file type, the embedded Java file (.mjava).

## The Generated Java for Embedded Java Files

Compared to a matrix, the generated Java for an embedded file is both more straightforward and more complex. The way it's more straight forward is that if you open up an embedded file and then also open up its generated Java file, you'll notice that the generated Java file is really just the same as the embedded file except that matrix instructions have been replaced with calls to generated matrix helper objects. But, the way the generated Java is more complex for an embedded file is that here, you there is also some secondary Java files that are generated. We'll get back to that. Let's start off by doing just what we had mentioned and open up an embedded Java file and compare it to its generated Java file.

### **MyEmbeddedClass.mjava**

```
package mywork;

import MATRIX mypackage::MyMatrix;


public class MyEmbeddedClass {

    public void printOutImagePath() {

        System.out.println("Here's the image path:");
        System.out.println(
            MyMatrix->IMAGE `logo`:>path
        );

    }

}
```

### **MyEmbeddedClass.java** (the generated Java file)

```
package mywork;

import mypackage.MyMatrix;

import net.unconventionalthinking.matrix.*;
import net.unconventionalthinking.lang.*;
import net.unconventionalthinking.matrix.symbols.*;

public class MyEmbeddedClass
    implements MyEmbeddedClass__Annotations {

    private final static AppControl_Base appControl =
        mypackage.AppControl.initializeApp();
    private final static ExecutionInfo executeInfo =
        appControl.executionInfo;

    public void printOutImagePath() {

        System.out.println("Here's the image path:");
        System.out.println(
            MyEmbeddedClass__MatrixWorker
                .accessMatrix_MyMatrix__17(executeInfo, this);
        );

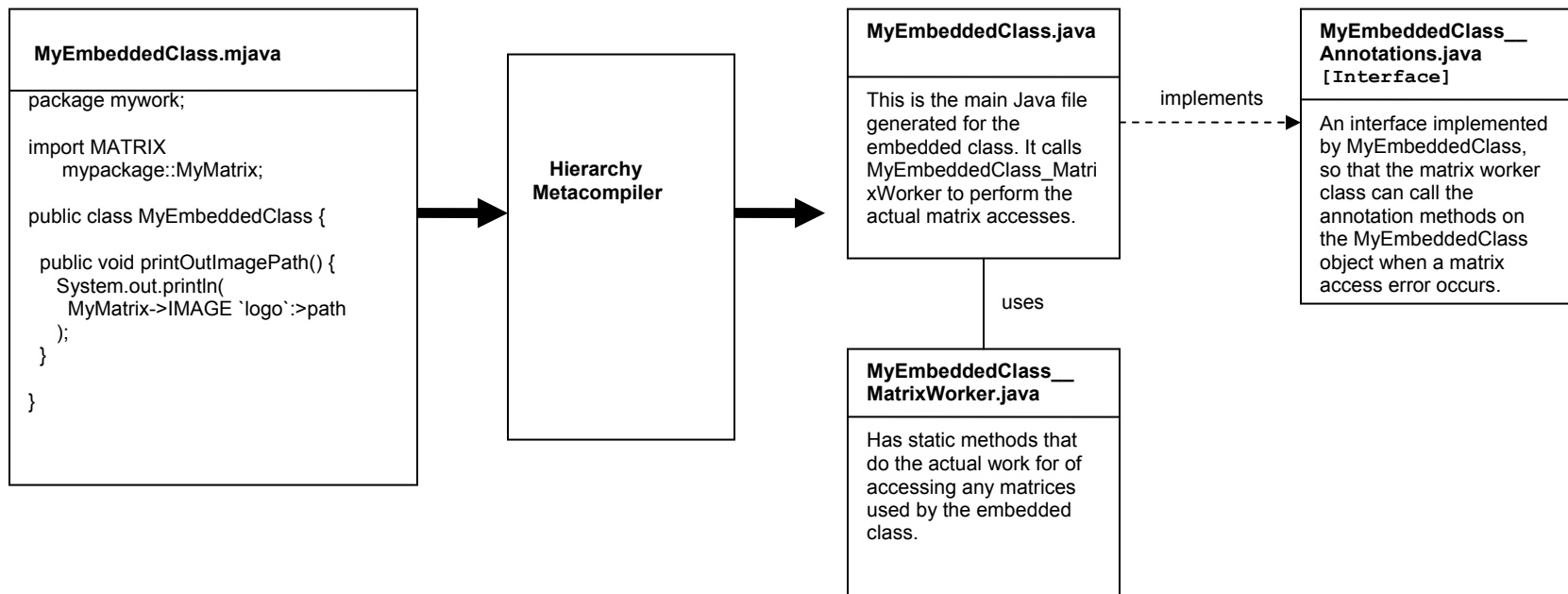
    }

}
```

All the code added by the metacompiler is in **bold text**. As you can see, there have been three main additions / modifications:

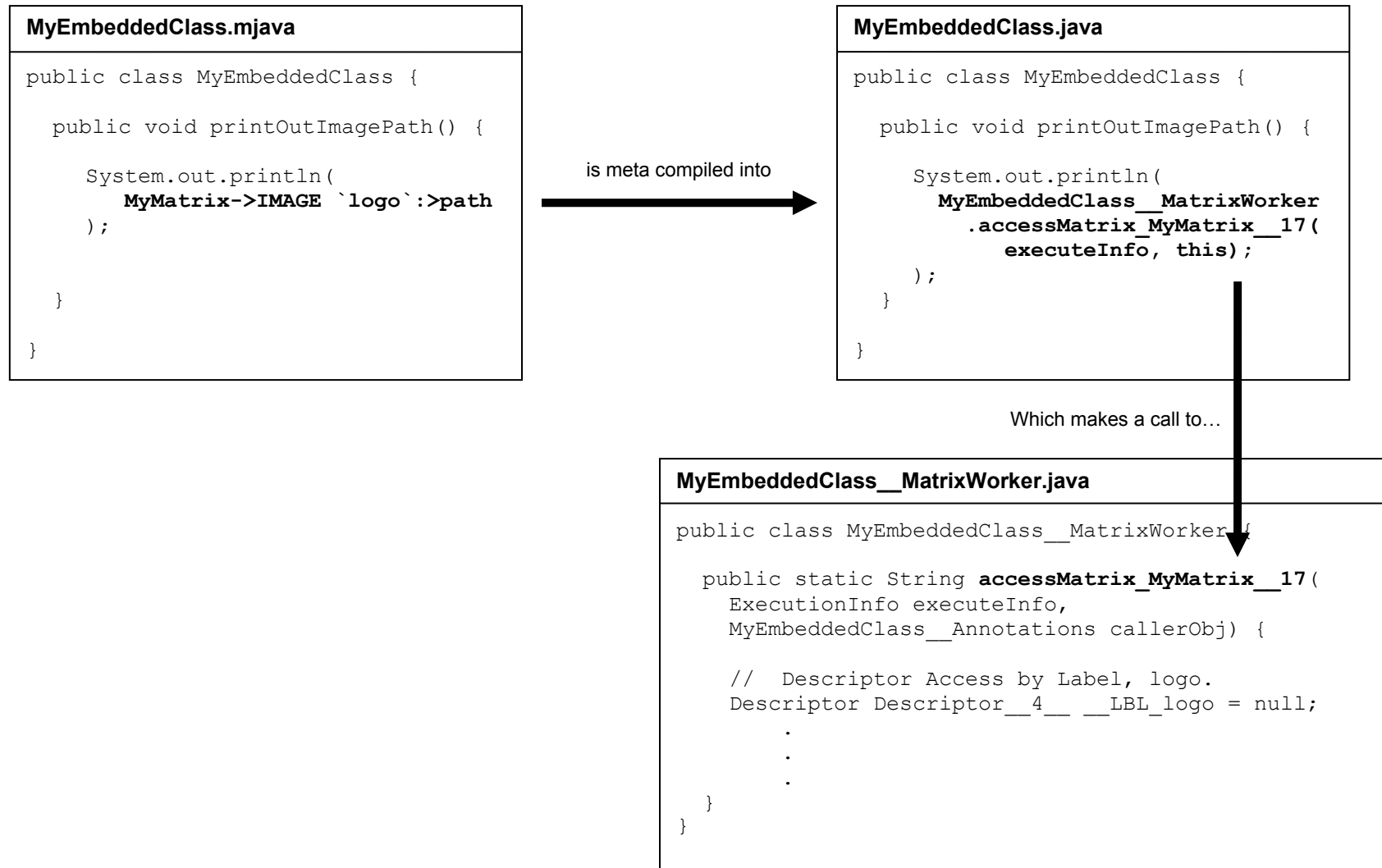
- The first is a few imports have been added to the class so it can do matrix access operations.
- The second is that two static objects, `appControl` and `executeInfo` have been added. The important object here is `appControl`. It is used by matrix applications as the central, runtime object for initializing and accessing matrices. The method call, `mypackage.AppControl.initializeApp()`, initializes all the matrix-related objects, calling the `construct()` methods for each of your matrices and schemas. As you have seen in the previous section, these `construct()` methods build the objects so they can be used in your application.
- The third is the conversion of the matrix access, `MyMatrix->IMAGE `logo`:>path`, into the static method call on the class' worker class, `MyEmbeddedClass__MatrixWorker.accessMatrix_MyMatrix__17()`. Here, the actual matrix access is done inside this static method, doing all the necessary work of descending through the matrix and finding the correct descriptor or field value to return back to the application. This worker class is another file that you'll need to open from time to time to figure out bugs in your embedded files. We'll take a look at this worker class shortly.

As we mentioned before, for each embedded file, there are some secondary classes that are generated. In total, there are three Java files for each embedded file:



The first file, `MyEmbeddedClass.java`, we've already talked about (it's the metacompiled version of the .mjava file from the previous example). The second file, `MyEmbeddedClass__MatrixWorker.java`, is a class that contains the methods that do the actual work of

accessing the matrices for MyEmbeddedClass. It contains a bunch of static methods, one for each matrix access that is done in MyEmbeddedClass. And so, in MyEmbeddedClass.java, each matrix access done in the MyEmbeddedClass.mjava file has now been replaced with a call to a static method in the MyEmbeddedClass\_\_MatrixWorker. This is probably easier to show than explain:



At this point, let's step through the code that does the matrix access in MyEmbeddedClass\_\_MatrixWorker. This code may look complex, but it's even maybe a bit simpler to follow than the matrix construction code:

```
package mywork;

import net.unconventionalthinking.matrix.*;
import net.unconventionalthinking.lang.*;
import net.unconventionalthinking.matrix.symbols.*;
```

```
public class MyEmbeddedClass__MatrixWorker {
```

```
    public static String accessMatrix_MyMatrix__17(ExecutionInfo executeInfo,
        MyEmbeddedClass__Annotations callerObj) {
```

First, always look for the comments. They will help guide you through the generated code. They occur at the start of a new descriptor access. Here, we're doing the access for the IMAGE 'logo' descriptor.

```
// Static Descriptor Access by Static Label, logo.
Descriptor Descriptor__244__STATIC__LBL_logo = null;
```

```
if (DescriptorUtilities.validDescriptors(executeInfo, mypackage.MyMatrix.matrix)) {
    try{
```

**\*\* Here's the important line \*\*** We're accessing **MyMatrix** and getting the child descriptor with the label, **logo**

```
        Descriptor__244__STATIC__LBL_logo =
            mypackage.MyMatrix.matrix.get_ChildDescriptor(executeInfo, AppSymbols_Label.logo);
    } catch (Exception e) {
        executeInfo.setErrorException(e);
    }
```

```
} else { // end of if (validDescriptors())
    executeInfo.addErrorInfo(ExecutionInfo.ErrorType.AccessedDescriptorThatWasNotFound);
}
```

This next block of code is the error handling code for the access. What this code does is:  
If the above access has a problem, the error is passed to the annotation handler.

```
// Error handling code
if (executeInfo.get_HadError()) {
    AnnotationParameters_AccessReturnType_OutParam accessReturnType_OutParam = new
        AnnotationParameters_AccessReturnType_OutParam();
```

Here, the annotation handler method is called. The important thing to note here is the **callerObject** is a **callback to the MyEmbeddedClass object**! Earlier in order to keep things simple, we didn't show that MyEmbeddedClass has the annotation handler methods on it. In fact, you should go to a sample project and open up a generated Java file for an embedded class. You'll see the methods for the annotations at the bottom.

```
Object annotateRetVal = callerObj.annotationHandler_4(executeInfo, false, null, 0, 0,
    AnnotationParameters.AccessType.Descriptor, accessReturnType_OutParam,
    mypackage.MyMatrix.matrix, mypackage.MyMatrix.matrix, null, true,
    executeInfo.getErrorException(), new Pair<Object, Object>(AppSymbols_Label.logo, null));

if (executeInfo.get_HadError() &&
    executeInfo.containsError(ExecutionInfo.ErrorType.AbortAccess))
    throw new ExceptRuntime_MatrixAccessError("Was trying the first access of the Matrix named
        mypackage.MyMatrix.matrix but had an error.");

if (accessReturnType_OutParam != null &&
    accessReturnType_OutParam.returnType == AnnotationParameters.AccessReturnType.accessorValue)
    Descriptor__244__STATIC__LBL_logo = (Descriptor)annotateRetVal;
else // accessReturnType_OutParam.returnType == AccessReturnType.accessValue)
    return (String)annotateRetVal;
}
```

We're starting the next access of the field, RequiredImagePath.

```
// Static Field Access by Static Descriptor Tag Name, RequiredImagePath.
if (DescriptorUtilities.validDescriptors(executeInfo, Descriptor__244__STATIC__LBL_logo)) {
    try{
        if (Descriptor__244__STATIC__LBL_logo.has_FieldSet()) {
            ** Here's the actual field access**
            1. We access the IMAGE 'logo' descriptor, Descriptor__244__STATIC__LBL_logo, and then call get_FieldSet() to
            access its fieldset object.
            2. Then, we upcast this object to the specialized fieldset object, FieldSetTuple__mypackage$CC_$Web$_S_$IMAGE.
            This specialized fieldset object contains methods generated just for working with the IMAGE Descriptor's fieldset.
            3. Lastly, we get the value of the Path field using the get_Path() call. It returns value as a string object.
            return ((mypackage.FieldSetTuple__mypackage$CC_$Web$_S_$IMAGE)
                Descriptor__244__STATIC__LBL_logo.get_FieldSet()).get_Path(executeInfo);
        } else {
            executeInfo.addErrorInfo(ExecutionInfo.ErrorType.AccessedFieldButNoFieldSetExists);
        }
    } catch (Exception e) {
        executeInfo.setErrorException(e);
    }
}
```

```

    }

} else { // end of if (validDescriptors())
    executeInfo.addErrorInfo(ExecutionInfo.ErrorType.AccessedDescriptorThatWasNotFound);
}

```

Error handling for fields work a little differently then what we just saw earlier for descriptors. Here, if we reach this point in the access, then we *must* have had an error. The reason is because if we had successfully accessed the field, we would have returned the value and not made to this point in the code. So now, process whatever error we had.

```

// Error handling code
AnnotationParameters_AccessReturnType_OutParam accessReturnType_OutParam = new
    AnnotationParameters_AccessReturnType_OutParam();

```

Here, the annotation handler method is called. This annotation handler works pretty much the same as the one we saw earlier for descriptors.

```

Object annotateRetVal = callerObj.annotationHandler_4(executeInfo, false, null, 2, 2,
    AnnotationParameters.AccessType.Field, accessReturnType_OutParam,
    mypackage.Test$__Matrix.matrix, Descriptor_244__STATIC__LBL_logo, null, true,
    executeInfo.getErrorException(), new Pair<Object, Object>(AppSymbols_Label.First$32$Name,
    null), new Pair<Object, Object>(AppSymbols_DescTag.mypackage$__CC$__Web$__CC$__IMAGE, null), new
    Pair<Object, Object>(AppSymbols_DescTagName.RequiredImagePath, null));

if (executeInfo.get_HadError() &&
    executeInfo.containsError(ExecutionInfo.ErrorType.AbortAccess))
    throw new ExceptRuntime_MatrixAccessError("Was trying the third access of the Matrix named
        mypackage.MyMatrix.matrix but had an error.");

return (String)annotateRetVal;

}
}

```

That's it for going through the code, yay! It wasn't easy, but I hope you see it's actually not that bad either. And, armed with this knowledge, you should be able to quickly solve any bug the metacompiler throws at you. In this next section, we'll go over the general steps you take to debug a problem.

## Debugging your .schema, .matrix and .mjava Files

You now know the metacompilation process, and how the generated Java code works. The next question is, how do we use this knowledge to actually debug errors found in your code during metacompilation? Here's the general debugging process you'd take to debug an error.

### Steps you'd take to debug a Metacompilation Error:

1. **Read the output of the metacompiler** – The metacompiler will try its best to return a useful error message to you. But, as you just saw in the previous section, the metacompiler may find the error in the metacode files (.matrix, .schema, .mjava) or in the generated Java files. Make sure you recognize which file type the error is in.
2. **Take different actions depending on where the error was found:**
  - a. **In the metacode** – if it's found in the metacode, the metacompiler should give you a pretty good error message, with a line number and position in the line of where the error is. These messages are usually pretty accurate and tell you exactly where and what the problem is, making them relatively easy to solve.
  - b. **In the generated Java files** – If it's found in one of the generated Java files, then locate the actual generated Java-file that was outputted to your metacompiler's `build_meta` directory (this path can be found in your metacompilation.properties file for the metacompiler). These types of errors messages are created by javac. Find the location of the error in the Java file, and use your new knowledge of how these generated files are structured to determine what the error is.

We've discussed what to do if you find a metacompilation error. But, how do you solve a runtime error in your matrix code? How do you debug it? We'll look at this next.

### Debugging Runtime Errors in your Matrix Code:

The general process is to treat the generated matrix code just like regular Java code, and set break points on generated Java code. Using your IDE, you can open up the generated java files and set break points in a matrix's `construct()` method, which is called during the initialization process. Or, you can set break points on different matrix accesses in your embedded code file's worker class (`<MyEmbeddedClass>__MatrixWorker.java`). And, since we've just gone over how these classes and methods work, you should be able to step through the code and determine what's going on.

There are a few useful things you should be comfortable with when setting breakpoints on a matrix application. We'll talk about these now.



## Useful tips for Debugging Runtime errors:

- **In your IDE's debugger, don't be afraid to inspect matrix and descriptor objects** – For instance, if there's a runtime error during the construction of a matrix, set a break point in the `construct()` method in the matrix's java file, and take a look at what values are being passed in during the construction of each Descriptor object. When you inspect a Descriptor object in the debugger of your IDE, there are a few important variables you should know. The most important variables on a descriptor are:
  - **descTag** – The descriptor tag for the descriptor. For example, it'd be set to `NEWS.STORY` for the descriptor:  

```
NEWS.STORY + `Save on Happy Dog Biscuits`: { "Save on Happy Dog Biscuits Today!", "We offer the lowest priced dog biscuits..." };
```
  - **label** – Continuing with the previous example, this variable would be set to:  

```
`Save on Happy Dog Biscuits`
```
  - **fieldSet** – holds the field set object whose variables are the fields for the descriptor. Continuing with the previous example, this descriptor's fieldset would have the values:  

```
Title -> "Save on Happy Dog Biscuits Today!"  
Content -> "We offer the lowest priced dog biscuits..."
```
  - **childDescriptors** – an array of references to the child descriptors for this descriptor
  - **parentDescriptor** – a reference to the parent descriptor for this descriptor

You use the `childDescriptors` and `parentDescriptor` variables to navigate around the matrix. And, you inspect the `fieldSet` object to see what field values the Descriptor object holds.

- **If you have a runtime error during initialization, when a matrix is constructed, set a break point in the matrix's `construct()` method** – We discussed the `construct()` method in the previous section, so you should have a good understanding of what's going on inside of it. Set a break point a couple of lines before the error, and use the IDE's debugger to inspect the different values of the descriptors to make sure they are correct.
  - **If an error occurs during the creation of a Descriptor object, double check all the values that are being passed in!** - This is the most common type of error you'll see in the `construct` method. Double check all the parameters being passed.
  - **If there's an error during the setting of field values, double check the types of the values being set** – This is also a very common error.
  - **The same techniques that you apply to debugging a matrix `construct()` method also applies to a schema's `construct()` method** – as mentioned in the previous section, schemas are matrices, so they're `construct()` methods are very similar.

- **A very useful object during debugging is the AppControl object** – It's the main control-panel object for your matrix application. This object is included as a static variable on every embedded Java file (.mjava). For instance, during debugging, a very common usage is to search the appControl object to find out if a matrix object is available.

**You can find the index of all available matrices at:**

- `AppControl.matrixControl.matrixIndex_<index organization type>`

**You can find the index of all available schemas at:**

- `AppControl.schemaControl.schemaIndex_<index organization type>`

The appControl object also has the `initializeApp()` methods, which are called during startup to construct all the matrix objects for your application. During the startup, if you have a problem with initialization of a matrix or schema object, trying setting break points in these initialization methods.

*It's highly encouraged that you open up a sample project, and try setting break points on the matrix code!* Debug the sample application and try inspecting a matrix and a schema object, and make sure you can easily navigate them (especially the matrix object). This is something you'll need to do from time to time.

At this point, you should be very comfortable with the concepts of how to debug your Matrix applications. In the last section of this chapter, we'll finish up by going over how to setup the metacompiler.

## Setting up the Metacompiler

The metacompiler has three dependant jar files it uses:

- Hierarchy jar (Hierarchy.jar)
- Apache Commons Collection jar (commons-collections-3.2.1.jar)
- Sable CC jar, version 3.2 (sablecc.jar).

To do the actual metacompilation of the meta-code files, we use this same Hierarchy.jar file we need during runtime – it's used for both metacompilation and at runtime. The reason for this is because you're also able to metacompile your files dynamically in code, at runtime using the Hierarchy jar.

So, when you start developing a Matrix application, you'll create your matrix meta-code files (so, your .schema, .matrix, and .mjava files) mixed in with your Java source files. The Java compiler can't do anything with these meta-codes files, but it makes sense to keep them there, since they are just enhanced Java files with extended syntax. And, all three types of files even have the Java package statement at the top identifying which package each file belongs to. Also, the Java compiler ignores any files in all the packages that don't have the .java extension, so you don't have to worry about matrix files somehow confusing the java compiler.

There are numerous ways to setup the metacompiler. One of the most common is to use a Java .properties file. We'll look at this next.

### The metacompiler's .properties file – hierarchy.properties

You set all the metacompiler settings in the hierarchy.properties file, and then you run the jar, passing the path to this file to the metacompiler. Here's a sample Hierarchy properties file:

```

# Properties file for a Sample Matrix project
# ** In paths, use forward slashes, '/', in both Unix and windows **

# Set the path to the Hierarchy.jar directory.
#hierarchy.jar.dir=/usr/local/hierarchy/lib
hierarchy.jar.dir=C:/java/lib/hierarchy/lib

# Set the path to the javac directory.
#javac.dir=/System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home/bin
javac.dir=C:/Java(x86)/jdk1.5.0_22/bin

# Set the class path used in All Phases of metacompilation
classpath.allphases=
# Set the class path used in just the Schema Phase of metacompilation
classpath.schema.phase=
# Set the class path used in just the Matrix Phase of metacompilation
classpath.matrix.phase=
# Set the class path used in just the Embedded Phase of metacompilation
classpath.embedded.phase=

# Set the location of base dir of the matrix source (in other words,
# the base dir of your project which has your matrix, schema, & embedded code files
source.metacode.basedir=src/java

# Set the package name that the Matrix application classes will be placed
matrix.appclasses.package=com.unconventional.matrix.app

# Set the path the the base directory where the intermediate, meta build files
# (generated java files) will be placed.
# Sometimes, it's useful to point this back to the 'source.metacode.basedir',
# so that the generated java files are placed back into the src directory.
build.meta.basedir=src/java
# Set the path the the base directory where the final class files will be placed.
build.final.basedir=matrix_build/build2_final

# Set whether to the final stage of compilation: Compiling the java files from the
# meta phase into the final class files. Default is true;
compileMetaFiles.into.finalClassFiles=true

# Set whether to clean the build meta and build final directories before
# metacompilation (true or false)
clean=false

```

*Example 40*

Now, we'll take a look the different settings,. Let's start with the first two:

```
# Set the path to the Hierarchy.jar directory.
hierarchy.jar.dir=C:/java/lib/hierarchy/lib

# Set the path to the javac directory.
javac.dir=C:/Java(x86)/jdk1.5.0_22/bin
```

The Hierarchy metacompiler needs the path to the directory where you're Hierarchy.jar and its dependent jars (such as the Apache commons-collections.jar and the Sable CC jar) are located. Also, it needs the path to the directory where your javac compiler executable is located. It uses these to do the meta-compilation.

The next group of settings deals with how to add any dependent jars to the classpath for your meta code files to use:

```
# Set the class path used in All Phases of metacompilation
classpath.allphases=
# Set the class path used in the Schema Phase of metacompilation
classpath.schema.phase=
# Set the class path used in the Matrix Phase of metacompilation
classpath.matrix.phase=
# Set the class path used in the Embedded Phase of metacompilation
classpath.embedded.phase=
```

Each of the main phases of the metacompilation may each use different class paths to compile their files. You can set each phase individually. And, to each of these class paths, the "allphases" classpath is added to it. Often, you'll probably just set the "allphases" classpath. And note, you don't have to add the Hierarchy.jar, the Apache commons-collection.jar, or the Sable CC jar to the classpath, these are added in for you.

Next, let's look at the paths for the directories for the input and output source-files.

```
# Set the location of base dir of the matrix source (in other words,
# the base dir of your project which has your matrix, schema, & embedded code files
source.metacode.basedir=src/java

# Set the package name that the Matrix application classes will be placed
matrix.appclasses.package=com.unconventional.matrix.app

# Set the path the the base directory where the intermediate, meta build files
# (generated java files) will be placed.
# Sometimes, it's useful to point this back to the 'source.metacode.basedir',
# so that the generated java files are placed back into the src directory.
build.meta.basedir=src/java
# Set the path the the base directory where the final class files will be placed.
build.final.basedir=matrix_build/build2_final
```

In the `source.metacode.basedir` setting, you provide the relative path (from where you'll be running the metacompiler) to the base directory of your source code, where all your Java files and Matrix meta-code files are located.

Next, in the `matrix.appclasses.package` setting, you provide the name of the package that all the Matrix-application java-files will be generated in. For each application (or for each jar) you create, the metacompiler

creates a package where it inserts some java files it uses to run the application. One of the most important is the `AppControl` class. This class is used during startup to initialize all the matrix, schema, and symbol objects in a matrix application. You can name this package anything you want, except the default package.

In the `build.meta.basedir` setting, you provide the relative path to the base directory where all the metacompiled Java files will be output. It is often useful to have these files generated right back into your source directory (if you notice in our example, we do this. Our `source.metacode.basedir` and our `build.meta.basedir` settings are the same). One common reason you'd want to do this is, after metacompilation, you can check to see if your generated matrix Java-files have any compilation errors in your IDE. More specifically, you'd create a project in your IDE that has a source directory that has both your Java and Matrix meta-code files. And, when you metacompile your matrix meta-code, the generated java files will be added right back into the same source directory. Now, you'll be able to see these generated files in your IDE, and then, using the IDE's Java syntax checker, you'll be able to see if there are any problems in the generated files. But, note, that if you do this, the generated java files get mixed in with your non-matrix java files, so things can get messy if you're not careful. For instance, if you start renaming you matrix files and then re-run the metacompiler, the previously generated java files with their old names will still be in there, and you'd have to remember to delete them – One solution to this is to try to keep you matrix and schema files in packages that aren't mixed with many other java files.

In the `build.final.basedir` setting, you specify the base directory where your compiled `.class` files will be added. This is an optional step, which you can turn off with the `compileMetaFiles.into.finalClassFiles` setting, as seen in this next code snippet:

```
# Set whether to the final stage of compilation: Compiling the java files from the
meta phase into the final class files. Default is true;
compileMetaFiles.into.finalClassFiles=true

# Set whether to clean the build meta and build final directories before
metacompilation (true or false)
clean=false
```

If the `clean` setting is true, before metacompilation, the metacompiler automatically deletes the files and directories that it generates, which are the directories of the `build.meta.basedir` and `build.final.basedir` settings. Be careful, because it's a blind delete and it simply deletes these two directories. But, the metacompiler won't allow you to clean if the `build.meta.basedir` equals the `source.metacode.basedir`, because then you'd be deleting all your source files.

Now we understand how to setup your metacompiler, how do you run it? You'd do this using the following command line command:

```
java -jar Hierarchy.jar -cp lib/sablecc.jar:lib/commons-collections-3.2.1.jar -
mpropfile <my hierarchy .properties filename>
```

The location where you run this command is important. The paths used in the `.properties` file are relative to the directory of the run location (or to the working directory).

Up to this point, we've discussed the metacompilation process, how to debug your matrix applications, and how to setup the metacompiler. There's one last thing to go over that deals with metacompilation, and that's how to setup the metacompiler to deal with Java dependencies. Let's move on to this now.

## Dealing with Java Dependencies

Your matrix file-types (.matrix, .schema, .mjava), can have different types of dependencies on .java files. This may not seem like an important subject, but it is pretty important, so in case you thought of skipping this last section, read on...

There are three ways that regular Java files can “depend on,” or “be depended on” by the matrix file types:

1. Matrix files can depend on Java files
2. Java files that depend on Matrix files
3. Java files that both “depend on” and “are depended on” by Matrix files.

We'll go over all three.

### Matrix files that depend on Java files

All the matrix files types (.matrix, .schema, .mjava) can have dependencies on Java files. For example, one very common thing to do is for a matrix to use a Java class to hold some of its constants. Typically, this constants class would hold values that have a lot of content, like a very long string of text, or constants that are shared globally:

```
package com.mychemicalcompany.website;

import com.mychemicalcompany.products.UltraChemToothpasteInfo;
import com.mychemicalcompany.departments.DepartmentInfo;

MATRIX Website.Content {

    PRODUCT.INFO +`UltraChem Toothpaste`: {
        // product description text, department name
        UltraChemToothpasteInfo.ProductDesc,
        DepartmentInfo.OralHygiene_DepartmentName
    }

    PRODUCT.INFO +`NeonBrite Facial Scrub`: {
        // product description text, department name
        "NeonBrite Facial Scrub will make your skin glow! Try it today!",
        DepartmentInfo.FacialProducts_DepartmentName
    }
    .
    .
    .
}
```

*Website\$\_\_ \$Content.matrix*

This matrix accesses the Java class, `UltraChemToothpasteInfo`, which is a *regular Java file* as it contains no matrix code! This class contains a variable that holds the actual text for product description field. Here is this class:

```

package com.mychemicalcompany.products;

public class UltraChemToothpasteInfo {
    // This constant hold the actual text for the product description field
    // for the Website.Content matrix.
    public static final String ProductDesc =
        "We found toothpaste works much " +
        "better when it's been treated by few of our highly " +
        "regarded and hi-tech chemicals. Fast, effective, and " +
        "the most powerful tooth solvent you can get on the market, " +
        "give UltraChem Toothpaste a try today! ...blah, blah, blah..." +
        .
        .
        .
        (this string goes on for many lines)
}

```

*UltraChemToothpasteInfo.java – a regular Java file!*

Here, we have a very long string of text that if it we added directly into the matrix file, may clutter it. So instead, we've moved this very long string out of the matrix into its own Java class. The metacompiler supports this type of dependency on Java files (we'll talk about how to setup this up in a moment).

There is a second Java class that the matrix accessed. It accesses `DepartmentInfo.OralHygiene_DepartmentName`, where this constant is a global value that is used through out the system. Here's what this second Java class looks like:

```

package com.mychemicalcompany.departments;

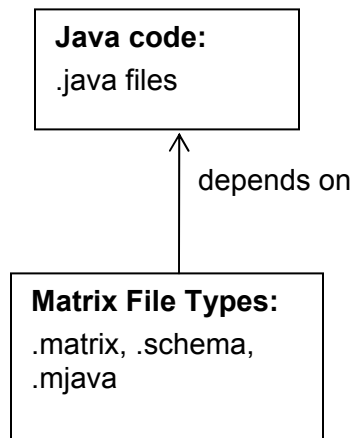
public class DepartmentInfo {
    // A global constant that is used throughout the system, including
    // our Website.Content matrix.
    public static final String OralHygiene_DepartmentName =
        "Oral Hygiene";
}

```

*DepartmentInfo.java*

These types of accesses of regular Java files by a matrix file type are perfectly valid. In these situations, we have matrix code that depends on Java code. Just to show this visually, here is this relationship:





On a practical level, for the .matrix, .schema, or .mjava files to depend on .java files, what needs to happen is the .java files that these file types depend on need to be compiled first! The metacompiler needs their .class files to be generated so that they can be linked to when the matrix file types are metacompiled. There are three ways to do this initial compilation of .java files:

**1. Have the metacompiler compile *all* the Java files initially, before metacompilation:**

The metacompiler will collect all the .java files in your src tree, and compile them with javac first.

The way you do this is simply setting a property in the metacompilation.properties file to true. The setting looks like this:

```
initiallyCompileAllJavaCode=true
```

Remember, if this setting is set to true, the metacompiler is going to compile *all* your .java files in your project *first*, before any of the matrix file types (.matrix, .schema, .mjava) are metacompiled. So, if you have .java files that need to be compiled *after* metacompilation (for instance, if a class in a .java file instantiates a class in one of the generated .mjava files), then using this setting will not work! It will cause a metacompilation error on any .java files that should not be initially compiled.

**2. Have the metacompiler metacompile a *select list* of Java files before metacompilation:**

Instead of compiling all the Java files initially, you pass in a specific list of Java classes that should be compiled before hand. This way, you can avoid any errors you might get with .java files that cannot be initially compiled in this way.

The way this is done is there is a second setting in the metacompilation.properties file where you can pass in a list of full class names that should be initially compiled:

```
# Note: this list actually must be on the same line, but is
# separated here to show the full example.
initiallyCompileJavaCode.SelectedClassesList =
    com.mychemicalcompany.products.UltraChemToothpasteInfo,
    com.mychemicalcompany.DepartmentInfo
```

### 3. You can manually compile the Java classes that need to be initially compiled (possibly in an Ant script):

If you'd like greater control over when your .java files get compiled, then you can use something like an Ant script to specifically compile any Java files that need to be initially compiled.

As we mentioned in the intro to this chapter, there is another direction for this type of dependency, where matrix file types depend on Java files. We'll take a look at this next.

#### Java files that depend on Matrix files

The other direction of Java dependency is where Java code depends on matrix code. Actually, what would be happening here is Java code depends on the *generated* Java code for a matrix file type. An example might be that you create some type of service that's uses matrices internally, but allows the external code that uses this service to be regular Java:

```
public class ProductDescServiceThatUsesMatricesInternally {

    public static String getProductDesc(String productName) {
        // Dynamically create a new label using the product name string
        Label productNameLabel = appControl
            .symbolControl.singleSymbol_Factory
            .createNew_Label (productName, false);

        // pick PRODUCT.INFO descriptor using a dynamic label that's passed in
        return Website.Content->PRODUCT.INFO [productNameLabel]:>Description;
    }

}
```

*ProductDescServiceThatUsesMatricesInternally.mjava*

And, here's a **regular** Java file that uses and depends on this embedded Java file.

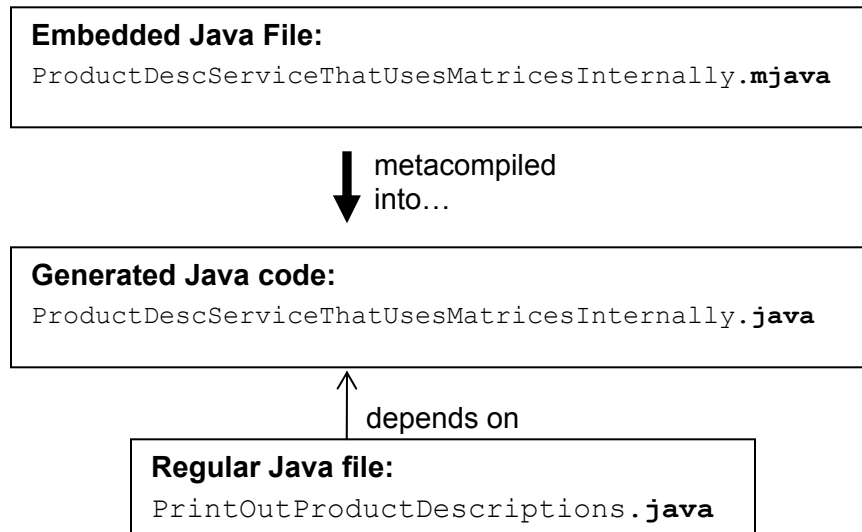
```
public class PrintOutProductDescriptions {

    public void printOutProductDesc(String productName) {
        System.out.println(
            // This REGULAR Java code calls a method on the embedded
            // Jaava Class!
            ProductDescServiceThatUsesMatricesInternally
                .getProductDesc (productName)
        );
    }

}
```

*PrintOutProductDescriptions.java* – a regular Java file that depends on an .mjava file

What's happening here is the regular Java class, `PrintOutProductDescriptions.java`, is depending on an embedded Java file, `ProductDescServiceThatUsesMatricesInternally.mjava`. But, as we mentioned earlier, what's really happening is this regular Java file will actually be depending on and accessing the generated Java file for the `.mjava` file:



In this situation, as you'd expected, you can't compile the regular Java file, `PrintOutProductDescriptions.java`, until you've metacompiled your matrix file types first! ... but, this shouldn't be a problem, because at the last stage, the metacompiler runs `javac` to compile all the generated and regular Java files. So, in this situation, you shouldn't need to do anything (note though, you can turn this final compilation of the Java file off in the `metacompilation.properties` file).

The last type of dependency is where Java files are *both* "depend on" and "are depended on" by Matrix files. Let's take a look at this next.

## Java files that both “depend on” and “are depended on” by Matrix files

Actually, Java files that both “depend on and be depended on by Matrix files types” only work with .mjava files! With the other two file types, .schema and .matrix, you can’t have Java files that are involved in this two way dependency. The reason for this only works with .mjava files is because they are metacompiled last, after both .schema files and .matrix files, so this means that they are *compiled* by javac last too. And since they are compiled last by javac, this also means they are compiled along with all the *regular* .java files for your project too. This means that .mjava files can both depend on and be depended on by Java files!

Not sure if this is clear, but what’s happening is the generated Java-files for .mjava files is compiled separately, in its own phase (you can see the phases of metacompilation by jumping back to an earlier section in this chapter on the “metacompilation process”). So anyways, what happens is .mjava files are *metacompiled* last and their .java files get generated. Now, the next thing the metacompiler should do is *compile* these newly generated Java files using javac. But, since this is the last step in metacompilation, we might as well compile *all* the other .java files too at the same time! (both regular Java files and the newly generated Java files for the .mjava files). And lastly, because they’re all being compiled together by javac, this means you can have two way dependencies between these file types.

Confusing?

Well, whether you understand the details, what this boils down to is for .mjava files, you’re allowed to combine the first two types of Java dependencies we just discussed. And, to get this type of two way dependency to work, just apply the instructions from those previous two types of dependencies.

## At this point, we done learning about metacompilation!

This chapter on metacompilation was a large one, so congratulations on getting through it! There were a lot of topics to go over, but they’re worth it. Once you get familiar with these topics, solving problems during metacompilation becomes fairly routine. We hope you find you now have a good understanding of how the metacompiler works.

## Chapter 8 – Starting your own Matrix Project

We'll show you two ways of starting you own project. The first is the easy way, which is simply to copy one of the sample projects and modify it. The second is the less easy way. It's to start a new project in your IDE, and then copy the necessary files and modifications from one of the samples into this new project.

### Method #1 – Copy One of the Sample Projects

The sample projects are good as templates to start new projects, because most are very simple, and don't have many unnecessary files. This is the easiest way to start a new project. The possible problem with this method is that there are only two types of sample projects: console applications and web. If the project you want to start doesn't fit into these two types, then you probably want to try the next method for starting a project, which is simply to start a new project in your IDE and copy in the files you need.

Let's return to explaining this method though. Here are the steps.

#### Starting your own console-application project or web project by copying one of the samples

1. Using a file manager, open `SampleProjects` in the `Hierarchy` directory for your IDE (Netbeans or Eclipse): `Hierarchy/SampleProjects/<Netbeans or Eclipse>`
2. If you're starting a new console-application project, copy the **MatrixSample\_Basic** project to a new location.  
If you're starting a new web project, copy the **MatrixWebSample\_Basic** project to a new location.
3. Open the project in your IDE, and use the IDE to rename the project
4. Now, jump to the introduction of this developer guide. Use the instructions in the intro to modify this copied project to work with the new name and location.

## Method #2 – Starting a New Project using your IDE

If you'd like to start an entirely new project in an IDE and add in the Hierarchy metacompiler settings manually, here are the steps. We'll provide instructions for both Netbeans and Eclipse. We'll start with the Netbeans instructions first.

### Starting a New Matrix Project in Netbeans – Non-Web Project

These instructions are for non-web projects! If you want to create a web project, see the next set of instructions.

1. Start a new project in Netbeans. It can be almost any type of Java project (except, for these instructions, no web projects. Just to repeat, the instructions for web projects are below).
2. Using your file browser, go to the sample projects directory, `/SampleProjects/Netbeans` and open the `MatrixSample_Basic` directory.
3. Copy the following files from the root directory of sample project to the root directory of the new project:
  - a. `buildMatrix.xml`
  - b. `hierarchy.properties`
4. In a text editor, open the `build.xml` files for the new project. This should be the Netbeans build script. Copy and paste the following text into the end of the xml file, right before the closing `</project>` tag:

```
<!-- This target calls the buildMatrix.xml's meta-compilation target.
      It meta-compiles the matrix files, generating java files. -->
<target name="-pre-compile">
  <subant target="metacompile">
    <fileset dir="." includes="buildMatrix.xml"/>
  </subant>
</target>

<!-- This target calls the buildMatrix.xml's clean target. It removes any
      of java files the generated during meta-compilation. -->
<target name="-post-clean">
  <subant target="clean">
    <fileset dir="." includes="buildMatrix.xml"/>
  </subant>
</target>
```

5. Just to test the new project can metacompile correctly, copy over the code from the sample project to your new project. You'll delete this code later!

From the `MatrixSample_Basic` project directory, copy the entire `samples` package in the directory: `src/samples`. Copy this directory into the `src` directory of your new project. Just to repeat, you're only using this code to test the metacompilation settings work. Again, you'll delete this code later!

6. Now, jump back to the introduction of this developer guide. Following the instructions for setting up the sample projects for Netbeans. These exact same instructions in the intro apply to setting up your new project! Then, try metacompiling your project to see that it works.
7. When you're done with the steps in the introduction, don't forget to delete the code you copied in to the project to test the build. This code is unnecessary!

## Starting a New Matrix Project in Netbeans – Web Project

These instructions are for web projects! If you want to create a non-web project, see the previous set of instructions

1. Start a web new project in Netbeans.
  - a. Use tomcat 6.0 as your server. If you don't have this installed, please go ahead and install it now.
  - b. Use Java 1.6
2. Using your file browser, go to the sample projects directory, `/SampleProjects/Netbeans`. Open the `MatrixWebSample_Basic` directory.
3. Copy the following files from the root directory of sample project to the root directory of the new project:
  - a. `buildJSP.xml`
  - b. `hierarchy_metacompilation.properties`
4. In a text editor, open the `build.xml` files for the new project. This should be the Netbeans build script. Copy and paste the following text into the end of the xml file, right before the closing `</project>` tag:

```
<!-- This target calls the buildMatrix.xml's meta-compilation target.
      It meta-compiles the matrix files, generating java files. -->
<target name="-pre-compile">
    <subant target="all">
        <fileset dir="." includes="buildJSP.xml"/>
    </subant>
</target>

<!-- This target calls the buildMatrix.xml's clean target. It removes any
      of java files the generated during meta-compilation. -->
<target name="-post-clean">
    <subant target="clean">
        <fileset dir="." includes="buildJSP.xml"/>
    </subant>
</target>
```

5. Just to test the new project can metacompile correctly, copy over the code from the sample project to your new project. You'll delete this code later!
  - a. From the `MatrixWebSample_Basic` project directory, copy all the child directories in the `src/java` to the same directory in your new project.
  - b. Also, copy over the two `.jspx` files in the `web` directory to corresponding directory in your new project:
    - i. `index-using-loopjspx`
    - ii. `indexjspx`

Just to repeat, you're only using this code to test the metacompilation settings work. You'll delete this code later!

6. You'll need to modify your `web.xml` (found in `web/WEB-INF`). Before we show what you need to do, just a quick explanation. When you run the `buildJSP.xml` file, it calls Apache Tomcat's JSP-compiler, `JspC`. `JspC` also adds configuration info on any servlets it generated to a file, `generated_web.xml`.

You need to modify your web.xml to include this generated xml. To do this, you'll need to make two additions:

**Addition #1** – Add this line *right after* the first line, `<?xml version="1.0" encoding="UTF-8"?>`:

```
<!-- This line is to create a reference to the "generated_web.xml" file, so
it can be included in the
    web.xml file -->
<!DOCTYPE generated-webxml [
    <!ENTITY generated-webxml SYSTEM "generated_web.xml">
]>
```

**Addition #2** – Add this xml code *right before* the closing tag, `</web-app>`:

```
<!-- We include the JspC-generated mappings here. -->
    &generated-webxml;
```

7. You also need to add this generated\_web.xml file to your project (JspC cannot actually create it). In the web/WEB-INF directory, create a new, empty file called generated\_web.xml.
8. Now, jump back to the introduction of this developer guide. Following the instructions for setting up the sample projects for Netbeans. These exact same instructions in the intro apply to setting up your new project! Then, try metacompiling your project to see that it works.
9. When you're done with the steps in the introduction, don't forget to delete the code you copied in to the project to test the build. This code is unnecessary!



## Starting a New Matrix Project in Eclipse – Non-Web Project

These instructions are for non-web projects! If you want to create a web project, see the next set of instructions.

1. Start a new project in Eclipse. It can be almost any type of Java project (except, for these instructions, no web projects. Not to get repetitive, but the instructions for web projects are below).

In the new project wizard, under the **Libraries** tab add the following jars:

- a. **Hierarchy.jar** – located in `Hierarchy/bin`
  - b. **commons-collections-3.2.1** – located in `Hierarchy/bin/lib`
  - c. **sablecc.jar** – (actually, not needed for most projects) located in `Hierarchy/bin/lib`
2. Using your file browser, go to the sample projects directory, `/SampleProjects/Eclipse` and open the `MatrixSample_Basic` directory.
  3. Copy the following files from the root directory of sample project to the root directory of the new project:
    - c. `buildMatrix.xml`
    - d. `hierarchy.properties`
  4. In Eclipse, turn off “Build Automatically.” Because the metacompilation process for these projects is more complicated than just the standard Java build-process, you don’t want to build every time you save. So, you can turn off build auto by unchecking the menu item, **Projects->Build Automatically**.
  5. You’ll need to setup up “Builders” for the project in Eclipse. These builders are fired off during when you do a “build” or a “clean” on your project. We can set them up to call specific targets in the `buildMatrix.xml` ant script which you just copied over. These ant targets perform different parts of the metacompilation process for you. The instructions look like a lot of steps, but they’re all pretty straightforward and are (honestly) not too bad.

To setup the builders:

- a. In Eclipse, right click on your project in the project explorer. Then select “properties.” In the project’s properties dialog, in the left nav, select “Builders”.
- b. In the Builder’s pane, click on the “New” button to create a new builder. A new dialog box will pop up asking you to select the type of builder to create. Select “Ant Builder,” and click OK.
- c. Another dialog should pop up called “Edit Configuration” which is used to setup the builder. First, enter the name of your builder, something like: “Build *<insert my project’s name>*”.

Setup the “Main” tab:

- d. Next, we’ll point the new builder to the `buildMatrix.xml` ant script. Under the “Main” tab, under “Buildfile,” click on the “Browse Workspace” button. Then, navigate to and select the `buildMatrix.xml` file you just copied into your project.
- e. Now, we’ll set the base directory for the ant script (this is the directory in which the script will run). Under “Base Directory,” click on the “Browse Workspace” button, and navigate to and select your current project.

Setup the “Refresh” tab:

- f. Next, you’ll setup the project’s refresh settings. This determines when the project will check for any newly generated files after the ant script has run, updating the files in the project explorer. First, click on the “Refresh” tab.
- g. Check the “Refresh resources on completion” checkbox.
- h. Select “The selected resource” radio-button. This will refresh just your project after each metacompilation.
- i. Make sure the “Recursively include sub-folders” check box is checked.

Setup the “Targets” tab:

- j. Now, we need to setup which targets in the ant script are called, and when in the build process these targets are fired off. Click on the “Targets” tab.
- k. In the “After a Clean” settings area, click “Set Targets.” In the “Set Targets” dialog box that pops up, you want to *uncheck* all the “targets to execute!” (The reason is because we’ll have the metacompiler do it’s cleaning not after a clean, but in the “During a Clean” step in the build process which we’ll setup below).
- l. In the “Manual Build” settings area, click “Set Targets.” In the “Set Targets” dialog box, under “targets to execute,” you want to check *just* the “metacompile” target. This is calling the buildMatrix.xml’s metacompile target, which will do the metacompilation for the project.
- m. In the “Auto Build” settings area, you want no targets to be executed. The reason is because the ant script is a relatively slower build than just regular Java builds, so we don’t want it to fire off automatically. This setting should already be set to no targets (it should say: “<Builder is not set to run for this build kind>”). But, if it has some targets set, then click “Set Targets,” and uncheck all the target settings.
- n. In the “During a clean” settings area, click “Set Targets.” In the “Set Targets” dialog box, under “targets to execute,” you want to check *just* the “clean” target. This is calling the buildMatrix.xml’s clean target, which will put the project back to a clean state after a metacompilation.
- o. Let’s sum up the settings for executing targets in the buildMatrix.xml:

<b>After a Clean</b>	NONE
<b>Manual Build</b>	Metacompile
<b>Auto Build</b>	NONE
<b>During a clean</b>	Clean

Last Steps:

- p. Finish creating the new builder by clicking the OK button.
  - q. This will return you to the Builders pane in the Properties dialog for your project. The last thing you need to do is move the new builder to the top of the list of builders. In the list of builders, select your builder and click the “Up” button until it’s at the top.
  - r. Click OK on the Properties dialog and you’re done! The way you know your new builder is working is when you do a build, in the console; you’ll see a lot of output from the metacompiler (hard to miss). You should also see a little output from the metacompiler when you do a clean as well. Even though you don’t have any code in your project, you may want to try building and cleaning now to test if it’s setup correctly.
6. Now, to further test the new project can metacompile correctly, let’s copy over some sample matrix code. We’ll copy over the code from the sample project to your new project. You’ll delete this later!
- From the `MatrixSample_Basic` project directory, copy the entire `samples` package in the directory: `src/samples`. Copy this directory into the `src` directory of your new project. Just to repeat, you’re only using this code to test the metacompilation settings work. Again, you’ll delete this code later!
- 7. Refresh the Eclipse project to see the new code in your IDE.
  - 8. Now, jump back to the introduction of this developer guide. Following the instructions for setting up the sample projects for Eclipse. These exact same instructions in the intro apply to setting up your new project! Then, try metacompiling your project to see that it works.
  - 9. When you’re done with the steps in the introduction, don’t forget to delete the code you copied in to the project to test the build. This code is unnecessary!

## Starting a New Matrix Project in Eclipse – Web Project

These instructions are for web projects! If you want to create a non-web project, see the previous set of instructions

\* There are actually a lot of steps to setup a web project in Eclipse. It's recommended that you be a little more careful as you perform these steps.

1. Start a new project in Eclipse.
  - a. As the project type, select **Web>Dynamic Web Project**
  - b. In the Dynamic Web Project wizard, you'll see a section where you can set the **Target Runtime:**
    - i. In the drop down, select **Apache Tomcat v6.0.**
    - ii. If this is unavailable, click on the **New Runtime** button and install Apache Tomcat v6.0.
    - iii. As the server's JRE, use JRE 1.6
  - c. Click on next until you hit the end of the wizard. Then, click on finished.
2. Using your file browser, go to the sample projects directory, `/SampleProjects/Netbeans`. Open the `MatrixWebSample_Basic` directory.
3. Copy the following files from the root directory of sample project to the root directory of the new project:
  - e. `buildJSP.xml`
  - f. `hierarchy_metacompilation.properties`
4. Copy the Hierarchy Jars to your `WebContent/WEB-INF/lib` directory:
  - a. **Hierarchy.jar** – located in `Hierarchy/bin`
  - b. **commons-collections-3.2.1** – located in `Hierarchy/bin/lib`
  - c. **sablecc.jar** – (actually, not needed for most projects) located in `Hierarchy/bin/lib`

Add the same jars to the project properties. Open up the properties for the project and the Properties dialog box should pop up. Then, select the Java Build Path settings on the left side. Then, select the Libraries tab. In the libraries tab, add these same jars to the project.

10. In Eclipse, turn off "Build Automatically." Because the metacompilation process for these projects is more complicated than just the standard Java build-process, you don't want to build every time you save. So, you can turn off build auto by unchecking the menu item, **Projects->Build Automatically**.
11. You'll need to setup up "Builders" for the project in Eclipse. These builders are fired off during when you do a "build" or a "clean" on your project. We can set them up to call specific targets in the `buildMatrix.xml` ant script which you just copied over. These ant targets perform different parts of the metacompilation process for you. The instructions look like a lot of steps, but they're all pretty straightforward and are (honestly) not too bad.

To setup the builders:

- a. In Eclipse, right click on your project in the project explorer. Then select "properties." In the project's properties dialog, in the left nav, select "Builders".
- b. In the Builder's pane, click on the "New" button to create a new builder. A new dialog box will pop up asking you to select the type of builder to create. Select "Ant Builder," and click OK.
- c. Another dialog should pop up called "Edit Configuration" which is used to setup the builder. First, enter the name of your builder, something like: "Build *<insert my project's name>*".

Setup the “Main” tab:

- d. Next, we’ll point the new builder to the `buildMatrix.xml` ant script. Under the “Main” tab, under “Buildfile,” click on the “Browse Workspace” button. Then, navigate to and select the `buildMatrix.xml` file you just copied into your project.
- e. Now, we’ll set the base directory for the ant script (this is the directory in which the script will run). Under “Base Directory,” click on the “Browse Workspace” button, and navigate to and select your current project.

Setup the “Refresh” tab:

- f. Next, you’ll setup the project’s refresh settings. This determines when the project will check for any newly generated files after the ant script has run, updating the files in the project explorer. First, click on the “Refresh” tab.
- g. Check the “Refresh resources on completion” checkbox.
- h. Select “The selected resource” radio-button. This will refresh just your project after each metacompilation.
- i. Make sure the “Recursively include sub-folders” check box is checked.

Setup the “Targets” tab:

- j. Now, we need to setup which targets in the ant script are called, and when in the build process these targets are fired off. Click on the “Targets” tab.
- k. In the “After a Clean” settings area, click “Set Targets.” In the “Set Targets” dialog box that pops up, you want to *uncheck* all the “targets to execute!” (The reason is because we’ll have the metacompiler do it’s cleaning not after a clean, but in the “During a Clean” step in the build process which we’ll setup below).
- l. In the “Manual Build” settings area, click “Set Targets.” In the “Set Targets” dialog box, under “targets to execute,” you want to check *just* the “metacompile” target. This is calling the `buildMatrix.xml`’s `metacompile` target, which will do the metacompilation for the project.
- m. In the “Auto Build” settings area, you want no targets to be executed. The reason is because the ant script is a relatively slower build than just regular Java builds, so we don’t want it to fire off automatically. This setting should already be set to no targets (it should say: “<Builder is not set to run for this build kind>”). But, if it has some targets set, then click “Set Targets,” and uncheck all the target settings.
- n. In the “During a clean” settings area, click “Set Targets.” In the “Set Targets” dialog box, under “targets to execute,” you want to check *just* the “clean” target. This is calling the `buildMatrix.xml`’s `clean` target, which will put the project back to a clean state after a metacompilation.

- o. Let's sum up the settings for executing targets in the buildMatrix.xml:

<b>After a Clean</b>	NONE
<b>Manual Build</b>	Metacompile
<b>Auto Build</b>	NONE
<b>During a clean</b>	Clean

Last Steps:

- p. Finish creating the new builder by clicking the OK button.
  - q. This will return you to the Builders pane in the Properties dialog for your project. The last thing you need to do is move the new builder to the top of the list of builders. In the list of builders, select your builder and click the "Up" button until it's at the top.
  - r. Click OK on the Properties dialog and you're done! The way you know your new builder is working is when you do a build, in the console; you'll see a lot of output from the metacompiler (hard to miss). You should also see a little output from the metacompiler when you do a clean as well. Even though you don't have any code in your project, you may want to try building and cleaning now to test if it's setup correctly.
5. Now, to further test the new project can metacompile correctly, let's copy over some sample matrix code. We'll copy over the code from the sample project to your new project. You'll delete this code later!
- a. From the `MatrixWebSample_Basic` project directory, copy all the child directories in the `src/java` to the same directory in your new project.
  - b. Also, copy over the two `.jspx` files in the `WebContent` directory to corresponding directory in your new project:
    - i. `index-using-loopjspx`
    - ii. `indexjspx`

Just to repeat, you're only using this code to test the metacompilation settings work. You'll delete this code later!

6. Refresh the Eclipse project to see the new code in your IDE.
7. You'll need to modify your `web.xml` (found in `web/WEB-INF`). Before we show what you need to do, just a quick explanation. When you run the `buildJSP.xml` file, it calls Apache Tomcat's JSP-compiler, JspC. JspC also adds configuration info on any servlets it generated to a file, `generated_web.xml`. You need to modify your `web.xml` to include this generated xml. To do this, you'll need to make two additions:

**Addition #1** – Add this line *right after* the first line, `<?xml version="1.0" encoding="UTF-8"?>`:

```
<!-- This line is to create a reference to the "generated_web.xml" file, so
it can be included in the
    web.xml file -->
<!DOCTYPE generated-webxml [
    <!ENTITY generated-webxml SYSTEM "generated_web.xml">
]>
```

**Addition #2** – Add this xml code *right before* the closing tag, `</web-app>`:

```
<!-- We include the JspC-generated mappings here. -->
    &generated-webxml;
```

8. You also need to add this `generated_web.xml` file to your project (JspC cannot actually create it). In the `web/WEB-INF` directory, create a new, empty file called `generated_web.xml`.
9. Now, jump back to the introduction of this developer guide. Following the instructions for setting up the sample projects for Eclipse. These same instructions in the intro apply to setting up your new project! Then, try metacompiling your project to see that it works.
10. When you're done with the steps in the introduction, don't forget to delete the code you copied in to the project to test the build. This code is unnecessary!

## Chapter 9 – Persistent Matrices with “Frictionless Persistence”

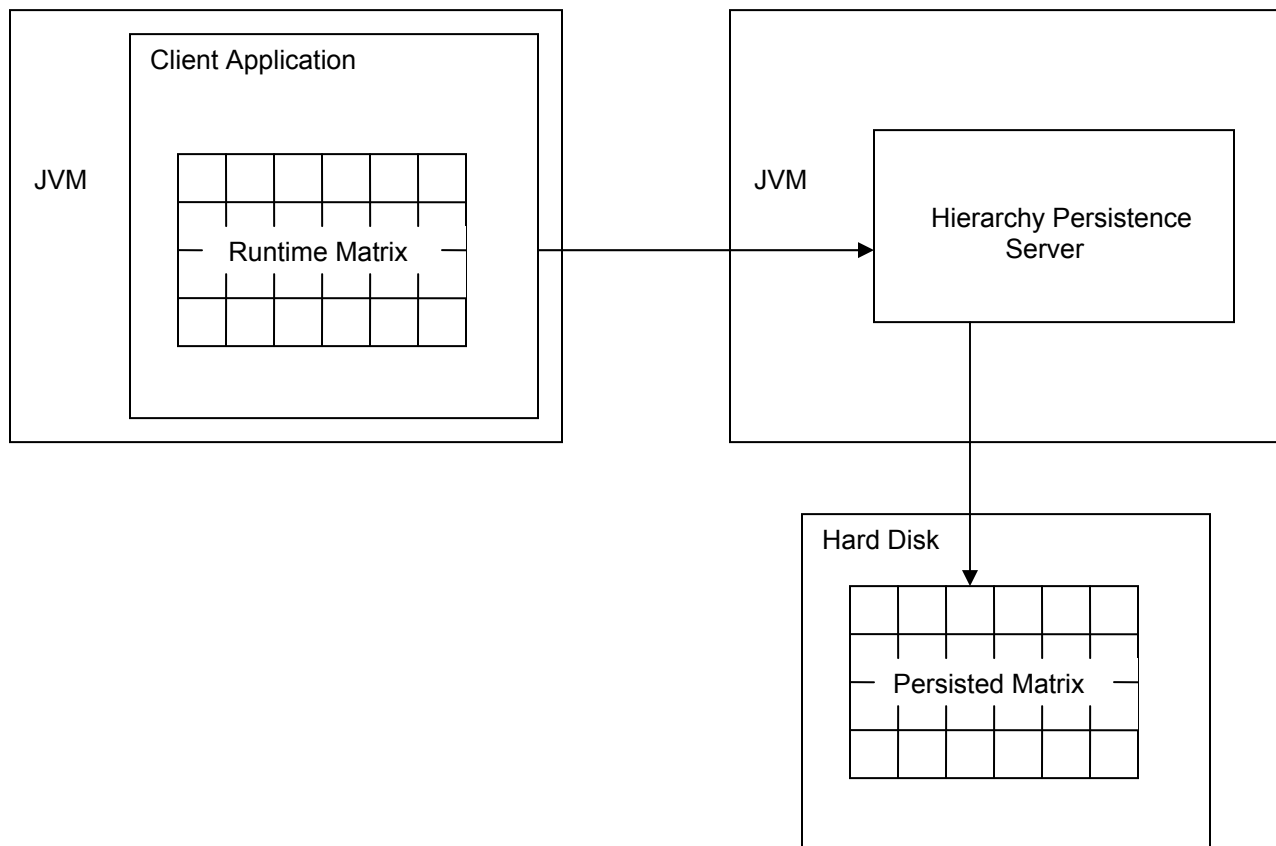
### What are Persistent Matrices?

As a developer, you’ve probably had to use persistence in one form or another over the years, whether it be using JDBC to connect to a database, or using an object-relational mapping framework like Hibernate which connects with a database. Persistent matrices work along the same lines. If an application is running with persistent matrices, any changes made to these matrices are sent to a persistence server. The persistence server adds these updates to a version of the matrices on disk. And, so as you might think, what this means is any updates an application makes to persistent matrices will survive when this application shuts down and restarts.

Based on this description, you might think this sounds a lot like Hibernate, but with matrices instead of Java objects mapping to a set of tables in a database. The answer to this is “yes” and “no.” The way it’s different is in two ways:

- First, matrix persistence is not object-relational mapping and does not map matrices to tables in a database. Instead, when matrices are persisted to disk by the persistence server, the server stores the matrices as actual matrices. In fact, matrix persistence does not even use a relational-database server.
- Second, as we just implied, applications do not connect to a relational-database server. Instead, they connect to a custom, matrix persistence-server that’s included with the Hierarchy runtime-library.

Here’s what an application with persisted matrices looks like:



Basically, the way this system works is:

- The application runs in its own JVM, separate from the persistence server, which runs in another JVM.
- Both the application and the server have a copy of the matrix. The running application has its matrix in RAM, while the server has its matrix on disk (actually, in this “preview” version of persistence, the server has two copies of each matrices, one in memory and one on the disk. The reason for this is because it was easier to implement this initial version of persistence this way. But, in the future, the server will for the most part have the matrices on disk).
- When the client application updates a matrix, this update is collected in an update log file. Once enough updates have been collected or a time limit has passed, this log file is sent to the persistence server. The server then processes the updates in this log file, adding them to the persisted matrix on disk.
  - This persistence strategy is similar to those found in other “in-memory” persistence products. When a client application does an update to persisted data, it doesn’t have to wait for the server to receive and process the update before proceeding. It can simply make the update on its own, runtime version of the data, and once the update gets written to the log, the client can proceed. The client knows the log file will eventually be sent over to the server and the server will add this update to the matrix.
  - Note though that a lot of the details of this process still need to be worked out, as this is only a “preview” version of persistence and is not ready for production use!

## How do I make my matrices, frictionlessly persistent?

Well, the first thing that we should reiterate is that you should not use persisted matrices in production! This is only a preview version of persistence, as the persistence server needs quite a bit of work to make it bullet proof and ready for prime-time usage. But, here are the general steps you’d take.

To setup persistence on a matrix you need do 4 things:

1. In your matrix, add the **MATRIXPROG.PERSISTENCE.SETTINGS** descriptor. It indicates that this matrix is persistent. You also have to add the schema for this descriptor to your project (you can find this schema in the sample persistence web project found in the samples directory). Then, in your matrix, you also need to import and USE this schema:

```
import SCHEMA net.unconventional.persistence::MATRIXPROG.PERSISTENCE;  
  
MATRIX WilliesPetstore.Content USES (MATRIXPROG.PERSISTENCE) {  
    MATRIXPROG.PERSISTENCE.SETTINGS: { :IsPersistent };  
    .  
    .  
    .  
}
```

**NOTE: In the near future, you will not have to use the descriptor to enable persistence. You’ll use a matrix setting to do this.**

2. Modify two configuration files, one for the client and one for server, setting up all the paths needed by each.
3. Start the Java rmi server pointing to the Hierarchy.jar
4. Start the Hierarchy Persistence Server



One of the design goals of matrix persistence that it be very easy to setup and use. We won't go over too much of the specifics, as Persistence is only for preview. But, you are highly encouraged at this point to try to run the persistence, sample-project found in the samples directory. It'll show you persistence in action. Look at the document, "Persistence Sample – README.txt" found in the PersistentMatrix\_WebSample directory. It'll give you detailed instructions on how to run persistence. Note though that this project currently only runs in Netbeans. Future versions will run in other IDE's as well.

... And, if you'd really are interested in trying out frictionless persistence on your own projects, contact the developer at:

[peter@unconventionalthinking.net](mailto:peter@unconventionalthinking.net)

He'll provide you with further details.

## Chapter 10 – Case Studies on When to Use Matrices

In this chapter, we'll take a look at some of the common usages of matrices. We'll present different cases when you might want to use matrices, and then discuss why they work for these situations. Let's dig right in.

### Case #1 – System Configuration File

One of the first situations that come to mind when you might want to use a matrix is as the config file for your application. You can use a matrix similar to how you'd use XML, which is to store all the configuration settings for your application.

Normally, if you did this in XML, you'd use something like JAXB or JAXP to load the XML settings into Java classes, and then use these Java classes in your application. This is pretty standard way to load config settings from XML and should be familiar to almost all devs that have used XML. So, when you try to use matrices with XML, your first thought maybe to simply replace XML with matrices and load the settings from the matrices into Java class. This is not recommended, but let's take a look at it.

In this example, we have a server that has a cache where it stores some of its data. The settings for the cache are collected in the following matrix:

```
package myapp.settings;

import SCHEMA myapp.schemas::Server.Config;

MATRIX Server.Settings USES (Server.Config) {

    CACHE.SETTINGS: {"/myserver/datacache"}
}

Server__$Settings.matrix
```

And, here's the schema used by this matrix:

```
package myapp.schemas;

SCHEMA Server.Config {

    DESCRIPTOR +:%CACHE.SETTINGS {
        FIELD.NAMES: { +:%CachePath };
        FIELD.TYPES: { :String };
    }
}

Server__$Config.schema
```

And, here's the part that is not recommended! This is an (embedded) class that access the Server.Settings matrix. More specifically, the class is a server configuration wrapper class. It wraps the matrix, loading the different settings into an instance of the class, which mirrors the values in the matrix.

\*Not recommended 😞

```
package myapp;

import MATRIX myapp.settings::Server.Settings;

public class ServerConfig {

    private String cachePath;

    public class ServerConfig() {
        cachePath = Server.Settings->CACHE.SETTINGS:>CachePath;
    }

    public String getCachePath() {
        return cachePath;
    }

}
```

#### ***ServerConfig.mjava***

At a design level, what we're doing is creating an abstraction over the matrix object so that we don't work with it directly. This seems like a good design principle as in the future, we may want to swap out matrices with some other technology, like xml or a database. But, the idea behind matrices is that they should be treated as if they were apart of the Java, as a first class citizen of the language. By treating it like a separate technology, you're often creating unnecessary classes over the matrices. Matrices should be integrated well enough into the Java language that you shouldn't need to do this. So, instead, try to use matrices directly. If we return to the example server, we can get rid of the ServerConfig.mjava class and use the matrix directly:

```
package myapp;

import MATRIX myapp.settings::Server.Settings;

public class MyServer {

    private String cachePath;

    public class init() {
        cachePath = Server.Settings->CACHE.SETTINGS:>CachePath;
    }

}
```

#### ***MyServer.mjava***

Much simpler and with a lot less code! To go into more detail, you should try to think of matrices as just another data structure in the Java language, on the same level as a class or an enum. Normally, if you create

an enum, you wouldn't also create a Java class over it to access it. An enum is a core part of the language and you'd use it directly. You should also try to treat matrices the same way.

In fact, one goal of matrices is to make it a form of universal meta data for Java. Right now, the data from a matrix is loaded from the matrix itself. But, in the future, matrices will support XML and relational databases. You'll be able to easily open up an XML document into a matrix and use the matrix access operators to navigate and manipulate XML data held in the matrix. You'll also be able to easily write this matrix out to XML again.

And, for databases, matrices will easily be able to pull data from a database in a similar manner to hibernate. These features are slated for release in the near future.

## Case #2 – N-Dimensional Architecture

### Collecting All the Content and Settings for an Application into a Matrix – Or what we'd like to call "N-Dimensional Architecture"

This is one of our personal favorite usages of matrices, and was one of the two inspirations for creating them. The basic idea is that we can pull all the high-level settings out of a system, and put them in their own layer, in to matrices. This is very similar to what happens when you use Spring and Dependency Injection to collect all the settings for your objects and put them into xml files, but takes these ideas much further, as you'll soon see (by the way, Spring and Dependency Injection were actually not inspirations for this architecture, it was just a convergence of similar ideas).

But, before we discuss NDA, let's discuss some of the existing architectural-concepts that NDA builds upon. If you have a decent understanding of MVC / 3-tier, this discussion should be fairly easy to follow.

For most business applications, we tend to use MVC or 3-tier style architectures. When we use this type of architecture, we are favoring the *behavioral* characteristics of a system, with less emphasis on its *abstraction* characteristics. This last statement may sound kind of vague, so let's break that down further.

When we use an MVC type architecture, the major questions we ask ourselves when designing a system are: *Is this particular object a model object? Or is it a view object?* We are placing the objects into architectural buckets each of which has a different **behavioral** characteristic of the system: *If this object deals with the user interface, then it's most likely a view object, so put it with the other view objects for you system. If the object deals with accessing the database, it's probably a model object, put it with the other model objects.*

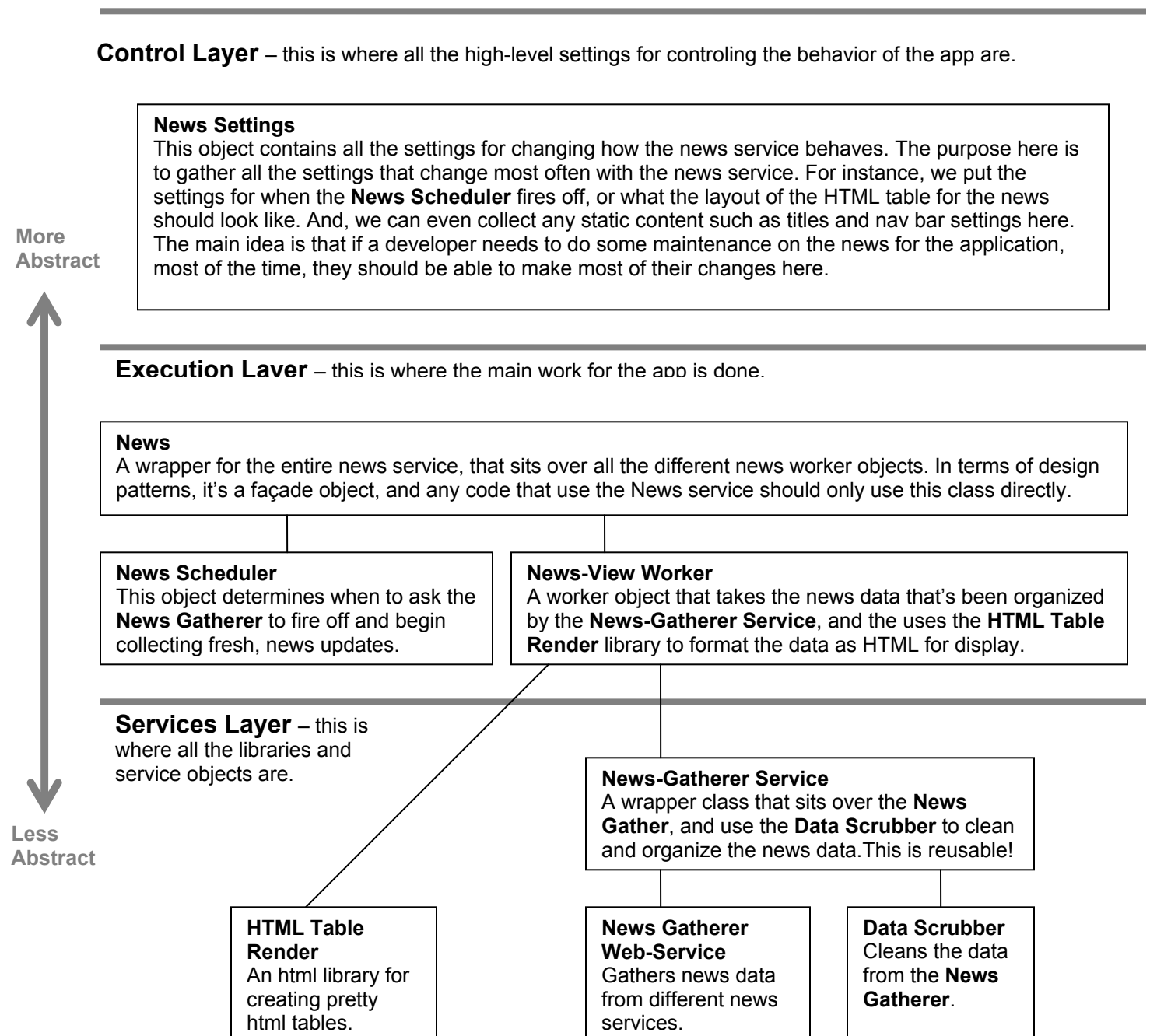
This is an extremely useful way of organizing a system, and the benefits of MVC to the discipline of software design cannot be overstated. But, we must remember that *abstraction* is also a very important way that we organize our systems. In fact, if we think about a computer, abstraction may be the most important way: The lowest layer of a computer is the hardware, with its different collection of chips and silicon to perform computations, graphics, and I/O. Then, above this, the hardware runs a mini operating-system, which is some form of BIOS that does the initial booting up of the different elements of the hardware. The BIOS also provides low-level API's that the main operating system can use to access the hardware. And, above this layer is the operating system itself, which provides many different services for starting and running applications. And, finally, above this are the applications, which use the API's of the operating system.

Most of us intuitively organize our own systems using abstraction as well, but we often don't make clear distinctions between the different layers. Typically, when we create a system, we ask ourselves questions like: *Is this object a low-level service that belongs in its own library? Or, does it belong at a level just above this, because it's a wrapper class that joins a bunch of different services together? Or, is it at an even higher level, using all the different service-objects to do the main work for the system?* When one organizes systems using these characteristics, one is organizing the system *by abstraction*.

But, we developers shouldn't limit ourselves to using *just* abstraction or *just* behavior (MVC) in our system architectures. We should consciously be organizing our systems by both at the same time! We call this type of architecture where you are organizing a system using multiple characteristics *simultaneously* an **N-Dimensional Architecture (NDA)**.

To see a NDA in action, let's design architecture for a sample, news-feed application. The way this news-feed application works is: based on a schedule, it periodically starts up and collects articles from different sites on the web. It then cleans the ads and pulls out the content, and then displays it to the user in an html table.

First, we'll start designing this system by organizing the different objects *by abstraction* (by the way, the following diagram may look a little daunting, but have no fear. We find it's easiest to just start by reading the description of the objects at the bottom layer and working your way up. It's actually not too bad):



*\*Typically easiest to read this diagram from the bottom up.*

Here, we've organized the system into three main layers. Let's describe each one (we'll start from the bottom layer as this lowest layer is typically the most familiar to most devs):

1. **Services Layer** – Basically, this layer contains our reusable libraries and web-services. This code is more *general purpose* and reusable through out the entire system (and possibly other systems as well). Examples of code in the Services Layer for our sample news-feed application would be: An in-house API for connect to the different media-site web-services, or a custom HTML-library for transforming the news articles grabbed from the different sites, converting them into a standard data-format.
2. **Execution Layer** – This is where the main work for the application is done. The Execution Layer *uses* the tools provided by the Services Layer to perform the features of the system. For our news-feed application, a couple examples would be the `NewsScheduler` component and the `NewsViewWorker`. These components handle the most important work for the application, and this layer is thought of as the “body” of our application.

But, there is another, even more important way to think about the Execution Layer. It is where the work *specific* to the system is performed. In the Services Layer, we placed our more *general-purpose* code there, like home-grown networking libraries or a custom data-scrubber API. But *here*, in the Execution Layer, we place the code that does the work that is *specific* just to this system.

3. **Control Layer** – In this highest-level layer, we pull out all the **settings** for how system behaves and all the **content** that is displayed to a user, grabbing this from throughout the entire application (*whatever* layer it is in). We put all these settings and content info together, placing them in high-level data-objects. For instance, for our news-feed application, some of the settings data we would grab would be: the settings for how to connect to each type of news website (web-service vs. JSON) and what data scrubber to use (XML vs. HTML documents), and the actual polling schedule for when the application should check the different websites/webservices for new articles.

We'll talk more about the Control Layer later on.

Organizing a system by abstraction is a very natural way of ordering a system. It's something we do intuitively, but should also be consciously done every time a developer designs a system. But, note, we are not replacing or throwing out MVC. In fact, the above architecture is *not* an NDA architecture yet. For that, we need to consider *both abstraction and behavior at the same time*. Here's the above system reorganized by both abstraction and behavior:

**Control Layer** – this is where all the high-level settings for controlling the behavior of the app are.

**News Settings**  
Often contains settings for all three cross sections.

More  
Abstract

**Execution Layer** – this is where the main work for the app is done.

**News** - A wrapper for the entire news service

**News-View Worker**  
A worker object that takes the news data that's been organized by the **News-Gatherer Service**, and uses the **HTML Table Render** library to format the data as HTML for display.

**News Scheduler**  
This object determines when to ask the **News Gatherer** to fire off

**Services Layer**

**HTML Table Render**  
An html library for creating pretty html tables.

**News-Gatherer Service**  
A wrapper class that sits over the **News Gatherer**, and use the **Data Scrubber** to clean and organize the news data.

**News Gatherer Web-Service**  
Gathers news data from different news services.

**Data Scrubber**  
Cleans the data from the **News Gatherer**.

**Presentation  
Cross-Section**

**Business Logic  
Cross-Section**

**Data  
Cross-Section**

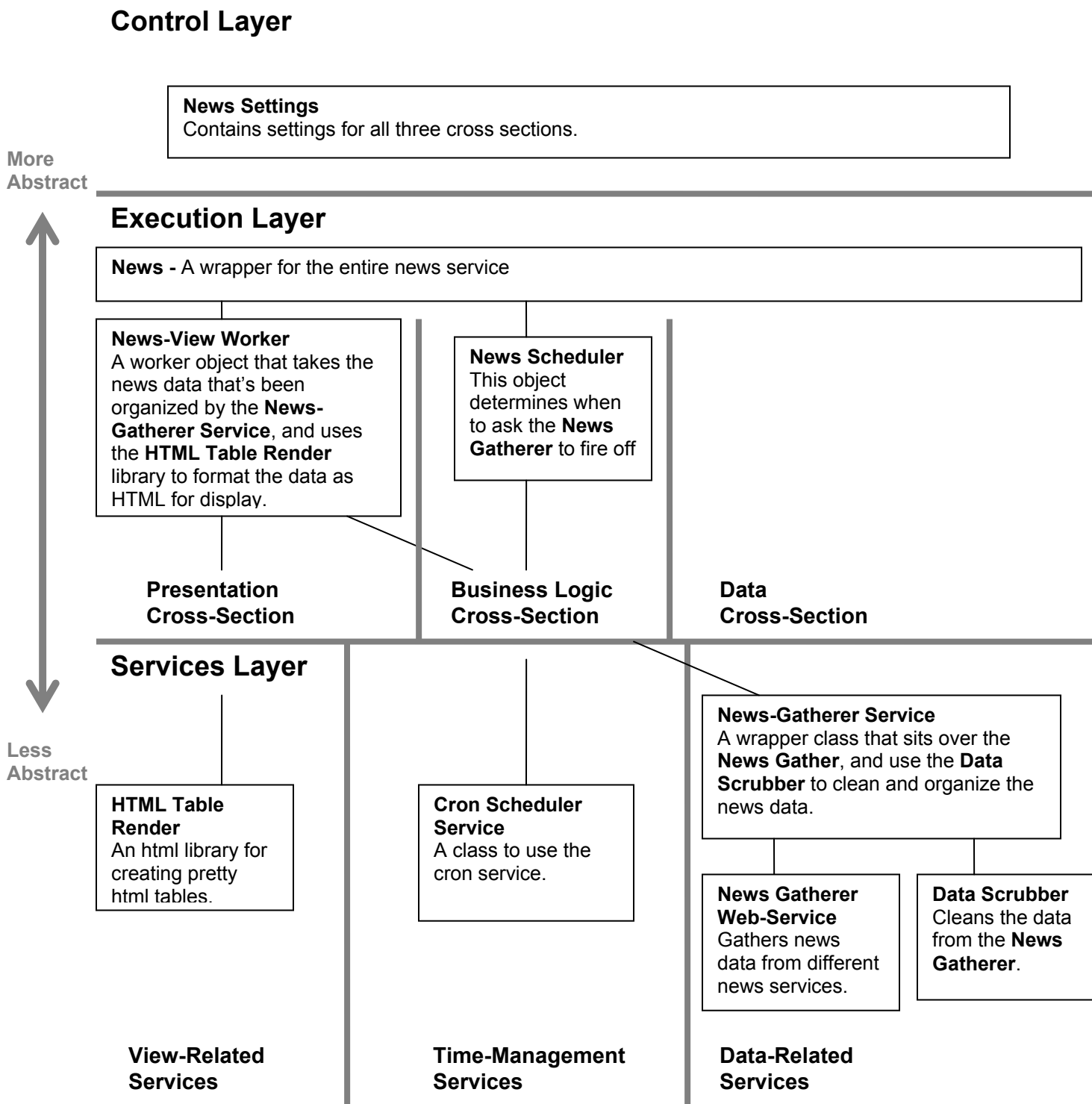
Less  
Abstract

Here, we've created another axis to order our system by. The vertical axis is still abstraction, but the new, horizontal axis orders the system *by behavior*, using an MVC style architecture. Notice that we are understanding the system two ways at the same time, by both abstraction and by behavior.



As you can see in this diagram, objects don't need to live in just one cross-section. Some objects span multiple or all cross sections. A good example is the **News** object that wraps all the other worker objects, or the **News Setting** object that contains settings for many different behaviors of the system.

And, we don't have to have the same horizontal axis for each layer. Each layer can have its own. In fact, it might make more sense to organize the system this way:



As we mentioned, one of the most important layers is the upper-most one, the **Control Layer**. To talk further about it, the idea is to put all the high-level info of our system *that changes the most* here. This way, devs will be able to look in one spot to do a majority of their changes, instead of having to search through the multiple parts of the system. An example of an object in the Control Layer for our news-feed application would be a data-source settings object. This object would be used to collect the different configurations for how to access each different news-website that the `NewsGathererService` connects to. Some settings that it might contain would be: the URL for the site and what format the articles were in (they could be XML generated from a news webservice, not just HTML). Now, anytime we had to connect to a new web site, a new data-source settings object could be quickly created for it.

But, moreover, for this new data-source, we would not only define the connection info, but all the other info relevant info for our system to work with this new website. For instance, the developer would probably need to create some type of data-scrubber to transform the news articles from the website's format to our own, internal data-format. Often in most systems, a dev would have to move to an entirely different part of the system to configure another component. But with an NDA architecture, we place the data-scrubber's config *with* its connection configuration. *And then*, on top of this, we also place all the view-type settings for this website here too, such as the settings for the html layout when a news article is displayed to our user. Unlike MVC architectures, we are grouping model, control and view settings together:

```
public class NewsDataSourceSettings {

    // CONNECTION SETTINGS:
    public String URL;
    public DataSourceType dataSourceType; // (see enum definition below)

    // DATA-SCRUBBER SETTINGS
    public DataScrubberType dataScrubberType; // (see enum definition below)
    public String dataScrubberTemplateFilename; // a template used by the
                                                // datascrubber to pull the content

    // HTML DISPLAY SETTINGS
    public DocumentDisplayLayout documentDisplayLayout; // (see enum below)
    public String documentBorderColor;
}

// This enum defines the different types of online news-feeds our
// application can connect to
public enum DataSourceType {
    HTML, WEBSERVICE, JSON
}

// This enum contains the names of the different types datascrubber
// objects that are available to us.
public enum DataScrubberType {
    // A custom library that was internally created that can be feed a
    // template to understand the layout of HTML websites (what is the
    // content area, nav bar, title bar...)
    TemplateBasedHtmlWebsiteScrubber,

    // This next scrubber library uses a neural net to learn the pattern
```

```

        // of the html layout to figure out the different areas.
        NeuralNetHtmlWebsiteScrubber,

        // This data scrubber cleans xml docs from webservices
        WebserviceXmlScrubber
    }

    // This enum defines the different types layouts an article can be
    // displayed in to our user
    public enum DocumentDisplayLayout {
        OneColumnFixedWidth, OneColumnVariableWidth,
        TwoColumnFixedWidth, TwoColumnVariableWidth
    }

```

Now, to actually define some settings objects, a dev would *new* instances of the objects, populating it with settings data:

```

// This is are the connections to the nytimes website.
NewsDataSourceSettings appSettings = new NewsDataSourceSettings(
    "http://www.nytimes.com", DataSourceType.HTML,
    DataScrubberType.TemplateBasedHtmlWebsiteScrubber, "nytimesTemplate.txt",
    DocumentDisplayLayout.OneColumnVariableWidth, "#FFFFFF");

```

What this boils down to is that when a dev has to add a new website to our system, he will often have to work with one or two objects to make a majority of his changes. These settings object become the primary source for his development task. And, the *overall* end-result is that the Control Layer becomes a *control panel* for the entire system, making it easy and efficient for devs to modify the application. If the Execution Layer is the body, the Control Layer is the brain, telling the body what to do.

...This might seem sacrilegious to most of us who have grown up on MVC to place model, view and controller settings together, but we have built many architectures using NDA, and it seems like the settings is the part of the system where religiously splitting things up by MVC is the least helpful. What we have found is that this control panel for the system acts basically as a user interface to the system architecture, where the users of this architecture are the developers. And, just like designing any other type of user interface, architects shouldn't organize them by the internal structure of the system, but by the *natural way a developer works* with the system.

As an analogy, if we were to create a product page for a website interface like Amazon.com, we wouldn't split up the different parts of the page across multiple sections of the website. Meaning that if a user is researching say the book, "How to Base Jump for Dummies," we wouldn't make the user, first, do a search for the product description of the book in the *Product* section of the site, and then, when the user wants to read the reviews have him redo his search in the *User-Review* section of the site, and lastly, when he looks up the price have him move again this time to the *Product-Pricing* section, redoing his search in this last section of the site. We would try to group the information he needs together on a single page that is well laid-out and easy to understand.

To talk more generally about the overall principles of designing *any* user interface (like for designing GUI's), it's probably safe to say, most *graphical* user-interfaces should *not* be organized by the internal structure of the system, but by principles of **Usability**. Usability is the study of "the ease of use and learnability of a human-made object." And, one of the main principles of Usability is *to design the interface around the tasks performed by the user*.

So, for the “*development* interface” to our news-feed system (meaning our settings objects<sup>1</sup>), these objects should also display their information to the user based on the ease-of-use of the *tasks performed by the devs* – more specifically, the tasks of maintaining and enhancing the system. And we just did this, as you just saw in our news-feed application. In our `NewsDataSourceSettings` object, instead of placing our connection settings for a news website in the config objects, then our data scrubber settings for the articles in a separate `datascrubberConfig` objects, and lastly our html display settings in another, separate `viewSettings` objects, we created one settings-object that contained all three! This object is now the primary source for a dev to perform this maintenance tasks for each data-source!

Also, it’s important to note that the idea is *not* to be rigid about putting everything for a dev task together, this can sometimes create settings objects that are huge and unwieldy. The idea is to group the settings as *logically and intuitively for the devs as possible!* Sometimes that means have multiple settings objects (very typical), sometimes that means having just one large one.

## How does Hierarchy fit into NDA?

For the implementation of the settings object in our news-feed application example, we used regular Java objects. But, actually, Hierarchy’s matrices are a better fit for holding this type information (this was one of the main inspirations for why Hierarchy matrices were created). They hold large sets of different type of data very easily and are easy to access from inside your Java programs.

To see this in action, let’s redo the news feed setting objects as a matrix:

```
MATRIX NewsDataSourceSettings USES (NewsDataSourceSchema) {  
  
    CONNECTION.SETTINGS: { "http://www.nytimes.com", :ConnectionType.Html };  
  
    DATASCRUBBER.SETTINGS: { :TemplateBasedHtmlWebsiteScrubber,  
        "config\nytimes\ntemplates\nnytimesTemplate.txt" };  
  
    HTMLDISPLAY.SETTINGS: { :OneColumnVariableWidth, :BorderColor.Blue };  
}
```

---

<sup>1</sup> The development interface also includes the front-most facing class definitions of the libraries most commonly used by the devs!

Pretty simple. And, here's the NewsDataSourceSchema schema definition:

```
SCHEMA NewsDataSourceSchema{

    DESCRIPTOR +:%CONNECTION.SETTINGS {
        FIELD.NAMES: { +:%URL, +:%ConnectionType };
        FIELD.TYPES: { : "String", :Symbol };
    }
    DESCRIPTOR +:%DATASCRUBBER.SETTINGS {
        FIELD.NAMES: { +:%DataScrubberType, +:%DataScrubberTemplate.FilePath};
        FIELD.TYPES: { :Symbol, : "String" };
    }
    DESCRIPTOR +:%HTMLDISPLAY.SETTINGS {
        FIELD.NAMES: { +:%DocumentDisplayLayout, +:%DocumentBorderColor };
        FIELD.TYPES: { :Symbol, :Symbol };
    }
}
```

As you can see, matrices are an extremely efficient way to collect all the settings of your system together. But, before we move on, let's take a look at a real world example of an NDA architecture. We'll take a look at a simplified version of the matrix used in the [unconventionalthinking.com](http://www.unconventionalthinking.com) website. First! Open up the Unconventional Thinking website in your browser (<http://www.unconventionalthinking.com>). Compare the information in the website as you read through the matrices in the following example. As we just mentioned, the website was created with heavy usage of matrices and should give you a look at using N-Dimensional Architecture in the real world:

```

MATRIX Unconventional.Content USES (com.unconventional.matrix.j2ee::Web.Content)  {

    `Home` {
        PAGE.INFO: { "home", 0, -1, "Home",
            "We build good software. We develop software products to help improve our world. We truly enjoy
            developing software that enhances your quality of life.",
            "", "", false };

        NEWS {

            NEWS.STORY: {"Hierarchy beta release",
                "The Hierarchy beta is ready for you to try!",
                "June 6<sup>th</sup>, 2011",
                "After years of development, the beta version of the Hierarchy Meta-Compiler for Java is ready
                for developers to try out!."
            };

            NEWS.STORY: {"N-Dimensional Architecture",
                "Research and Development on N-Dimensional Architecture Yields New Meta-Compiler for Java",
                "December 1<sup>st</sup>, 2009",
                "We have been researching theories on system architecture for over four years and are nearly
                ready to show the results of our countless days working late into the night..."
            };

        }
    }

    `Products` {
        PAGE.INFO: { "products", 1, -1, "Products",
            "We truly enjoy creating good software. Many applications are slow, crash or have poorly designed
            features. Bad applications make users sad. Building software that is powerful and easy to use, and
            reliable and responsive takes a lot of hard work, but we feel the greatest satisfaction in seeing
            our users happy. ",
            "", "", false};

        NEWS {
            NEWS.STORY: {
                "Matrix", "Hierarchy for Matrix-Programming ", null,
                "We are excited to finally let users try what we've been working on for the past 4 years: " +
                "the Hierarchy Beta is ready for Java developers to try out!..."
            };
        }
    }
}

```

```

`Contact` {
  PAGE.INFO: { "contactus", 5, -1, "Contact Us",
    "Please feel free to contact us",
    "", "", false};

  // { +:Person_ID, +:Name, +:Title, +:Role, +:Email, +:Description }
  PEOPLE {
    PERSON.INFO: {"info", "Information", null, null, "info@unconventionalthinking.net",
      "If you like to speak with someone at Unconventional Thinking..."
    };
    PERSON.INFO: {"sales", "Sales", null, null, "sales@unconventionalthinking.net",
      "If you like to speak with someone from Sales, please feel free to send an email here."
    };
    PERSON.INFO: {"careers",
      "Careers",
      null, null,
      "careers@unconventionalthinking.net",
      "If you interested in a career here at Unconventional Thinking..."
    };
    PERSON.INFO: {"support",
      "Support", null, null, "support@unconventionalthinking.net",
      "If you have a support question on one of our products..."
    };
    PERSON.INFO: {"location",
      "Location", null, null, null,
      "Unconventional Thinking is located in beautiful Ann Arbor, Michigan..."
    };
  }
}

```

### ***Unconventional\$\_\_Content.matrix***

In this matrix, we've collected all the content for the site. Again, if you haven't done so already, take a look at the real, unconventionalthinking.com website and compare the generated html with this matrix. You should see where different matrix elements line up with the generated content.

This next file is the jsp file that uses this matrix. Jsp files that have embedded matrix code use the **.mjsp file extension!** So, this file is called index.mjsp.

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@page import="MATRIX com.unconventional.web::Unconventional.Content"%>

<%

    // This next method call looks at the request objects parameters and determines which page we're on. Then,
    // it grabs the corresponding descriptor for the page from the Unconventional.Content matrix
    DESCRIPTOR<Unconventional.Content->ITEM> curr_PageItemDesc = determineCurrPageItemDescriptor(request);
%>

<table cellpadding=0 cellspacing=0 border=0 bordercolor=black width=532>
<tr>
    <td ><div style="width:500px">

        <font style='font-size:15px;font-weight:bold'>
            Title%"><br>
        </font>
        <div style="margin-top:17px">
            BlurbText)%">
                style="margin-top:3px" border="0">
        </div>

        <% if (curr_PageItemDesc.label == `Home`) { %>
            <div style="margin-top:47px; ">
                <br>
                <div style="margin-top:25px">

                    <%
                        for (DESCRIPTOR<Unconventional.Content->ITEM->NEWS->NEWS.STORY> newsItemDesc :
                            curr_PageItemDesc->NEWS->NEWS.STORY{*}) {
                    %>
                        <%=newsItemDesc:>Title%><br>
                        <%=newsItemDesc:>DateText%><br>
                        <%=newsItemDesc:>Story_Blurb%><br>
                    <% } %>

                </div>
            </div>
        </div>
    </td>
</tr>
</table>

```



```

<% } else if (curr_PageItemDesc.label == `Products`) { %>
  <div style="margin-top:62px">
    <div style="margin-top:12px">

      <%
        for (DESCRIPTOR<com.unconventional.web::Unconventional.Content->ITEM->NEWS->NEWS.STORY>
          newsItemDesc : curr_PageItemDesc->NEWS->NEWS.STORY{*}) {
      %>
        <%=newsItemDesc:>Title%><br>
        <%=newsItemDesc:>Story_Blurb%><br>
      <% } %>

    </div>

  </div>
</div>

```

```

<% } else if (curr_PageItemDesc.label == `Contact`) { %>
  <div style="margin-top:34px; width:340px">
    <table cellpadding=0 cellspacing=0 border=0 bordercolor=black>

      <%
        for (DESCRIPTOR<Unconventional.Content->ITEM->PEOPLE->PERSON.INFO> personInfoDesc :
          curr_PageItemDesc->PEOPLE->PERSON.INFO{*}) {
      %>
        <tr>
          <td valign=top>
            Name%>"><br>
              <a class="main-body-email" href="mailto:<%=personInfoDesc:>Email%>">
                <%=personInfoDesc:>Email%></a>
            </div>
            <%}%>
          </td>
          <td width=100% valign=top></td>
        </tr>
        <tr>
          <td colspan="2">
            <div>
              <%=personInfoDesc:>Description%>
            </div>
          </td>
        </tr>
      </table>

```

```

        <% } %>

    </table>
</div>

<% } %>

</div>
</td>
</tr>
</table>

```

***index.mjsp***

Here, in the [unconventionalthinking.com](http://unconventionalthinking.com) website, we're pulling out all the content and settings and placing them into the `Unconventional.Content` matrix. Then, we use this matrix to generate all the information for the site in the `index.mjsp` page. This is a typical usage of N-Dimensional architecture to separate elements from your system that change the most with those that change the least.

And, for those that are interested, on the next page is a simplified version of the schema used by this matrix.

```

package com.unconventional.matrix.j2ee;

SCHEMA Web.Content {

    DESCRIPTOR +:%PAGE.INFO {

        FIELD.NAMES: { +:%Name_NoSpaces, +:%Page_ID, +:%Page_childID, +:%Title, +:%BlurbText, +:%LNav_Href,
            +:%LNav_Image, +:%LNav_IsActive, +:%ContentAreaFormat };

        FIELD.DESC: { "Name of the Page (no spaces)", "The page id (int)",
            "The child page id (optional)(int)", "Page Title",
            "Short, descriptive text about the page", "The left nav Href",
            "The left nav image name", "The left nav, is active field",
            "The formatting of the content area. Symbol values are: :Normal and :Wide" };

        FIELD.TYPES: { :String, :int, :int, :String, :String, :String, :String,
            :boolean, :Symbol };

        FIELD.DEFAULTS: { null, null, -1, null, null, null, null, true, +:Normal };

    }

    DESCRIPTOR +:%NEWS {
        DESCRIPTOR +:%NEWS.STORY {
            FIELD.NAMES: { +:%NewsItem_ID, +:%Title, +:%DateText, +:%Story_Blurb, +:%Story_Text };
            FIELD.DESC: { "Id of news item", "News item description", "Date", "The short blurb about the story",
                "The text of the story." };
            FIELD.TYPES: { :String, :String, :String, :String, :String };
            FIELD.DEFAULTS: { null, null, null, null, null };

        }
    }

    DESCRIPTOR +:%PEOPLE {
        DESCRIPTOR +:%PERSON.INFO {
            FIELD.NAMES: { +:%Person_ID, +:%Name, +:%Title, +:%Role, +:%Email, +:%Description };
            FIELD.TYPES: { :String, :String, :String, :String, :String, :String };
            FIELD.DEFAULTS: { null, null, null, null, null, null };

        }
    }

}

```

**Web\$\_Content.schema**

As we keep mentioning, putting all the content and settings for the entire website together makes it easy to maintain the site. And to see this in action for this site, the most common change we (the maintainers of the [unconventionalthinking.com](http://unconventionalthinking.com) & [projecthierarchy.org](http://projecthierarchy.org) websites) find we have to perform is modifying or adding new content. It is a very simple task for us to do so with matrices: simply find the correct location to put or modify the content and add it in. We find for most of our maintenance tasks on these sites, we have to make almost all our changes in the matrices, with the occasional need to modify one or two other files.

In addition, we also find that we've had to write much less code for the same subsystems as compared to traditional MVC style systems. We think this may have to do with the fact that a lot of the organizational structure of MVC puts a lot of small amounts of information in lower level classes when they really belong at a higher level.

Now, to wrap things up, let's go into more detail about the benefits of N-Dimensional Architecture.

## Benefits of N-Dimensional Architecture

### Developer-Task Automation

As we looked at the code for the Unconventional Thinking website, we see that the creation of many of the elements in the website have been *automated*. What we mean by automation is, for example, how the news event info that's displayed on the home page for the site is dynamically generated from the matrix data (as opposed to being hard coded). So that now, if we need to add another news story to the home page, we simply need to add a new NEWS.STORY descriptor to the matrix, and the code for the site will then dynamically generate this new news story.

So, to define this more generally: **Developer-Task Automation** is taking a development task that a programmer typically would have to perform through multiple changes directly to the code and providing some type of interface (like a matrix) so that these changes can now be done in one spot, without actually having to write any (or little) code, just by changing some of the settings. And, a system architecture that automates developer tasks is called an **Automated Architecture**.

There are a lot of benefits to automation. Maintaining an automated system-architecture can be often much easier. If a developer needs to add or modify a new element to an automated component, you don't need to change any of the code, just the data that's used by the component. But, the cons of using automation are: often, developers need to write a good amount of additional code as compared to hard coding the content in a static component. But, with matrices, automated code is much more efficient to write.

It's worth mentioning that in other, non-NDA systems, automation is actually sometimes done, but normally using a DB or XML file. So, if we were to do the news feed with these techniques, we'd create a database table for the news info, and create a matching News model-object in a hibernate layer. And then, create application code that would use this model object to query the DB for the news info and format the results in a news feed. That's quite a bit of work. But, with matrices, automation is much simpler with much less code. You don't need a database, a database connection, or any model objects. The info for the dynamically generated content is simple stored in a matrix, which is a native data structure in our system.

Creating automated components when practical is often a good idea, but with Hierarchy and matrices, automation becomes much more easy to do. In our systems, we have found we create automated components three to four times more than we normally do. Especially for websites, because of their content-driven nature, practically the entire site can be designed around them (the [unconventionalthinking.com](http://unconventionalthinking.com) website is nearly entirely an automated system-architecture).

## High-Performance Automation

And, compared with a database, automated components in Hierarchy have a couple of orders of magnitude better performance. The reason is because Hierarchy stores matrices in RAM, while a database access requires a trip over the network to retrieve data. Accessing a matrix is comparable to accessing an array or a hash, they're extremely light weight.

In a real world usage, again the Unconventional Thinking website, we found the generation of dynamic elements was so fast, we didn't need to add any caching of generated HTML. This makes using Hierarchy for automation ideal on many different levels, on ease of use and on performance.

## Developer-Oriented Architecture

Using N-Dimensional Architecture tends to make systems **Developer-Oriented**. Developer-Oriented architecture is based on the Use Cases (*user tasks*) found in User-Oriented Software Development. It's basically the same idea. We previously mentioned this a few pages ago, but to build on what we said: In Use-Cases, the designers of an application try to view the system through the eyes of the user, and create tasks or "Use-Cases" for common actions the user would perform with the system. Developer-Oriented Architecture is the same thing. Developers are now users of the system, and a software architect tries to determine the maintenance and enhancement tasks that a future developer will need to perform. The architect is trying to see the architecture through the eyes of future developers.

N-Dimensional Architecture (NDA) is a Developer-Oriented Architecture. There are two main reasons why this is true:

1. NDA architectures put the elements that change the most at the top of the system and collect them together. So now, developers don't need to search across the entire system to make changes. In NDA, we are designing the system around the needs of the developers, and not just to meet the needs of end users.
2. NDA architectures tend to have **Development Interfaces**. Because we are ordering the system by abstraction, components and layers tend to get wrapped in classes that act as main interfaces into the component (or layer). These development interfaces are used by developers to work with the components, and so developers do not directly access any of the classes behind the development interface. An example of a development interface can be seen a few pages ago in the sample N-Dimensional Architectural diagram. In the Execution layer, there is a high-level class called "News." This News class wraps the entire news component. This is the main class that developers will use to work with it, and they won't directly access any of the objects behind this class. This is really just encapsulation, but on a larger scale.

And, as we previously saw, the high-level settings objects are another type of developer interface. The developer uses these to change the behavior of the system.

Also, since we think of this development interface as an actual interface for developers to work with the system, one of the goals of an architect is to create these development interfaces with a high degree of ease of use or what we call **architectural usability**. We want this interface to be as easy for developers to use as possible.

One last small note. Notice that this News wrapper class is following the façade design-pattern. The Façade design-pattern is used a great deal in Developer-Oriented Architectures.

This was a summary explanation of some pretty detailed concepts, but we hope you get a feel for why N-Dimensional Architecture can be beneficial to your systems.

## Case #3 – Universal Data Definition

In large, business applications, data elements (like customers info) are often used in many places throughout the system and must be defined in multiple places. For instance, for a user-registration page of a website, there is typically a last-name field. Think of the places this data element must be defined for just this simple page:

1. The HTML form:

```
<input name='lastName' max='25'>
```

2. The client-side Javascript validation code:

```
var lastName = form.lastName;
checkNameString(lastName, 25); // Of course, there are often libraries/frameworks
that                               // do validation, but they still need the value and
                                   // settings passed to them.
```

3. The server-side code for the JSP/JSF registration page:

```
public class userRegPage {
    String lastName;
    String firstName;
    .
    .
}
```

4. The DB model objects:

```
public class userDAO{
    String lastName;
    String firstName;
    .
    .
}
```

5. In the DB itself:

```
Table User
  lastName VARCHAR;
  firstName VARCHAR;
  .
  .
```

For a developer to modify an existing data-element (for instance, if he needs to change the Last Name field's max character length from 18 to 25), he may need to move to five different parts of the system to make a relatively simple change. Not only is this inefficient, it's error prone as the dev is likely to miss a necessary modification in one of the locations.

We can solve this problem of having a data element defined in multiple places using the concepts of NDA. Specifically, we'll apply the technique of grouping all the developer's most commonly used settings together along with **Automated Architecture** (refer to the previous case on NDA for more info on this concept).

The basic idea is: For each data-element, collect all its data-type definition, placing all this together. This can be placed into arrays of objects, or more ideally, into a matrix (again, see the “Main/Learn” section on the project Hierarchy website for more info on matrices).

This matrix becomes the **universal data-definition** (or universal scheme) for the entire application, in all its different usages. So, for our user-registration page example, the matrix for this info would look like this:

```
package com.williespetstore;

MATRIX WebForm.Registration USES (Web.Form, Database) {

    `First Name` {
        FORM.REQUIRED: { +:IsRequired };
        FORM.CONTROL.TEXTBOX { 25 };
        HELP.TEXT: { "Please enter in your first name" };
        // Field names: Table, ColumnName, Type
        DB.COLUMN: { +:Customer, +:First_Name, :String };
    }
    `Last Name` {
        FORM.REQUIRED: { +:IsRequired };
        FORM.INPUT.TEXTBOX { 25 };
        HELP.TEXT: { "Please enter in your last name" };
        DB.COLUMN: { +:Customer, +:Last_Name, :String };
    }
    `Gender` {
        FORM.REQUIRED: { +:IsRequired };
        FORM.CONTROL.RADIOBOX { "GenderRadioBox" } {
            RADIO.ITEM { "Male", "Male", :Checked };
            RADIO.ITEM { "Female", "Female" };
        }
        HELP.TEXT: { "Please select your gender" };
        DB.COLUMN: { :Customer, +:Gender, :int };
    }
    `Address 1` {
        FORM.REQUIRED: { +:IsRequired };
        FORM.CONTROL.TEXTBOX { 60 };
        HELP.TEXT: { "Please enter in your home address" };
        DB.COLUMN: { +:Customer, +:Address1, :String };
    }
    .
    .
    .
}
```

*\* Notice that in this matrix, we're using ITEM descriptors as the container for each field's information (if you're unfamiliar with the ITEM descriptor, see "Chapter 5 – Matrices"). ITEM descriptor are good for grouping related information together.*

Each field in the registration form has four types of information:

- Is this a required field?
- What type of form field it is.
- Help text
- The database column information that corresponds to each data-element

We can now use this matrix to generate a multi-page web form using a servlet or JSP. And also, we can use it to generate the insert query's SQL to store this information in the database. We can create this code in .mjava files (or .mjsp files if you're using JSP's. And for those that have not learned about Hierarchy, .mjava and .mjsp's are special files for the Hierarchy metacompiler that simply have extended Java syntax for working with matrix objects). Here's an .mjsp that uses this matrix to auto-generate the web form:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@page import="MATRIX com.williespetstore::WebForm.Registration"%>

<form>
  <%
    for (DESCRIPTOR<WebForm.Registration->ITEM> formField_Item:
      WebForm.Registration->ITEM{*}) {

      if (formField_Item->FORM.CONTROL:>ControlType == :TextBox) {
        %>
        <input type='text' name='<%=formField_Item.label%>'
          size='<%=formField_Item->FORM.CONTROL:>size'%>' />
        <%

      } else if (formField_Item->FORM.CONTROL:>ControlType == :RadioBox) {
        %>
        <div>
        <%

          for (DESCRIPTOR<WebForm.Registration->ITEM->FORM.CONTROL->RADIOBOX->
            RADIO.ITEM radio_Desc :
              formField_Item->FORM.CONTROL->RADIOBOX->RADIO.ITEM{*}) {
            %>
            <input type='radio' name='<%=radio_Desc:>name>'
              value='<%=radio_Desc:>value'%>' />
            <%
          }
          %>
        </div>
        <%

      } else if (...) {
        // Add other else-if statements to generate the rest of the
        // form elements from the matrix.
        :
        :
      }

    }
  %>
</form>
```



This code generates the user-registration form by looping over all the data elements in the WebForm matrix. The benefits of using matrices in this way (as a source for the “universal data definition” for your application), is now, all the definition information for each data-item isn’t scattered across the application, where some of it is in the database layer in Hibernate objects, some in JSP’s, and others in the Javascript. Instead, it’s all been collected into one location, the matrix. And you can use this information to more easily process this form in a more automated fashion. It is an example of **Automated Architecture**, because the elements of the form aren’t hard coded, but instead, generated using a description of what the form should look like. In fact, in the future, we’ll do most of the automation for you, as we plan to release libraries for both form generation and databases queries. And because of this automation and the fact that you’ve collected the settings together, the form should be much easier to maintain. For instance, if we need to add a new field, you don’t have to search through the system for all the different places we need to add information for this field, you simply need to add it to one place, the matrix.

One important note about the matrix: notice that the above Universal Data-Definition matrix was extended to mix *settings and content information* along with the data-definition info. This may seem like a mistake, as most of us are taught in architectures like MVC to “separate the concerns,” but this was actually on purpose. In our experience using NDA architecture, we have come to realize that at the high-level, strict separation of concerns (*often robotically done using MVC*) does not always lead to the best architectures. When introduced, MVC was a revolutionary architecture and its usage to *some degree* in our business applications will almost always be beneficial. But, for developers who work with the system, its maintainability and ease of understanding the architecture is often not ideal as it may separate elements that might best be kept together (MVC is often used as a one-size-fits all type of architecture, where in many cases the blind application of its principles isn’t ideal for the situation).

What our experience has found is that the mid-level (manager type objects) and lower level (libraries and services) objects often benefit by a stricter separation of concerns, but the high-level, application-specific content and settings is often better kept together, with the content organized more logically by how the developers naturally would group this information! This *may* mean a MVC style grouping, but often it doesn’t, and instead, results in an organization scheme designed around *how the developer sees and works with the system*. One common scheme is to keep conceptually related settings together: for instance, the matrix above is organizing its info around the data elements used in the system, and is actually grouping data and view settings together!

This may seem like sacrilege, but having more natural groupings of our settings leads to easier understanding of the system and greater efficiency in making changes. On top of this, as many of us agree, most applications *don’t* under go the drastic future enhancements we developers often plan for (like allowing for the adding of new views), so one of the main reasons necessitating the separation is never used. Also, since matrices are very light-weight objects, if, in the future, we need to refactor them and pull settings out, it is relatively easy to do so.

*But, on the flip side*, in many situations, it is actually not a good idea to have all the settings and content in one matrix! In many situations, for instance where the system is large, mixing info like this could lead to future problems. In these situations, it is advisable to split the settings into its separate matrices based on expected future development. For instance, if this code actually *will* have multiple views (for example, it will support both HTML and PDF versions of the pages), then splitting off the html settings into its own matrix may make sense.

The main point is, how your organize this highest layer of settings of your systems should not be done using a blanket, MVC approach. Each design requires its own thought about the correct scheme and balance to use.

## Final Words

### ***Congratulations on finishing this developer guide!***

There's a lot of information to go through, so we appreciate your taking the interest to read through it all! And, if you want to read more, see the Community / Articles section of our website for further docs on Hierarchy, NDA and advanced concepts of these two: [http://projecthierarchy.org/index.jsp?PAGE\\_ID=3&CHILDPAGE\\_ID=4](http://projecthierarchy.org/index.jsp?PAGE_ID=3&CHILDPAGE_ID=4)

And, we've got one further request of you. Hierarchy is currently a beta product, so *please* send in any bugs or problems you find with using the metacompiler or the documentation. We are especially interested in making Hierarchy as easy to use as possible, so if you ever find that you spent more than a few minutes trying to get something to work in the metacompiler, shoot us an email, we'd greatly appreciate the feedback!

Also, if you run into any problems, please check the FAQ document. It contains a list of solutions to many of the tasks and problems you might run across.