

COMP 557 – Tutorial 0: Eclipse, Java, MINtools

TA: Emmanuel Piuze

McGill University

Friday, September 16th, 2011

Introduction

This tutorial will help you get started for assignment 0 and subsequent assignments. Let me know if something is not clear, if you find a mistake, if you would like to add something to this tutorial, or if you simply find it useful! Topics that will be covered include the following:

1. Java programming language
 - Refresher (syntax, operators, loops, constructors, etc.)
 - Basic data structures and operations (lists, hash tables, etc.)
 - Vecmath package (vectors, points, matrices, quaternions and operations on them)
2. Eclipse open development platform
 - Setting up a Java workspace (project builder, window placement, etc.)
 - Quick navigation (jump-to's, class methods, fields, etc.)
 - Code refactoring
 - Hot-code debugging (runtime modifications and effects)
3. Swing and MINtools
 - Basic Java Swing elements and syntax (JButton, JLabel, JSlider, etc.)
 - Parameter types (boolean, integer, double) and listeners
 - UI elements (control frames, collapsible panels, etc.)
 - Interactivity (KeyAdapter, MouseMotionListener, keyPressed, mouseMoved, etc.)

Contents

1	Java	3
1.1	History	3
1.2	Commenting Your Code	3
1.3	Primitive data types	3
1.4	Operators	4
1.5	Methods	4
1.6	The Main Method	5
1.7	Rounding	6
1.8	Logic	6
1.9	Loops	7
1.10	Classes	8
1.11	JRE Convenience Classes	9
1.11.1	Lists	9
1.12	Vecmath package	10
1.12.1	Points and Vectors	10
1.12.2	Matrices	11
1.12.3	Quaternions	12
2	Eclipse	12
2.1	History	12
2.2	Verifying if Java is Installed	13
2.3	Installation	13
2.4	Configuring JREs and JDK Compiler Compliance	13
2.5	Workspace Management	14
2.6	Perspectives, Views and Packages	14
2.7	Creating a Java Project	14
2.8	Creating a Folder, Package, Class, Interface, Enum or Others	15
2.9	Must-Read: Tips and Tricks	15
2.10	Code refactoring	16
2.11	Debugging and Hot Code Replace	17
2.12	JVM Heap Space and Out of Memory Exceptions	18
3	Swing and MINtools	19
3.1	Master of the UI: Creating The EasyViewer	19
3.2	Building your UI	20
3.2.1	The getControls() method	20
3.2.2	CollapsiblePanel	20
3.3	Parameters	21
3.3.1	Constructors	21
3.3.2	Controls	21
3.3.3	Listeners	22
3.4	Keyboard and Mouse Input	23

1 Java

This part of the tutorial will give you a brief Java refresher. Most concepts are not explained thoroughly so I assume that you have had some experience with Java at some point in your life. If it's not the case, I highly recommend that you dig through some examples in any Java reference book you can find on the web or at the Schulich Library.

1.1 History

Java is an object-oriented programming language developed by Sun Microsystems (now Oracle) in 1995. Its syntax is similar to C and C++ but has a simpler object model and runs in a protected virtual machine, thus exposing less low-level facilities. There is no such thing as a pointer in Java: everything is passed by reference, except for primitive types (e.g. `int`, `double`) and the special `String` class. Memory allocation is also being taken care of automatically, as well as deallocation (garbage collection). Java applications are multiplatform in the sense that once compiled to bytecode they can run on any machine that has a Java Virtual Machine (JVM) installed.

1.2 Commenting Your Code

First and foremost, the most important and most neglected part of any programming language, the **comments**. In Java, comments can be single-line or multiline, and can optionally be (although recommended by standard coding guidelines) of two different flavours. The first flavour, for both single- and multi-line comments, should be used inside methods for describing blocks of code or for commenting code out. The second flavour should be used everywhere else, for example when describing the signature of a method or a class.

```
1 // 1. This is a single-line comment of the first flavour.
2
3 // 2. Commenting a block of code using the first flavour in multi-line.
4 /*
5  double z = Random.nextDouble();
6  double y = z * z;
7  */
8
9 /** 3. This is a single-line comment of the second flavour */
10
11 /**
12  * 4. This is a multi-line comment of the second flavour.
13  * Note the double asterisk at the top.
14  * This method adds up two numbers together.
15  * @param x the first number
16  * @param y the second number
17  */
18 public int add(int x, int y) {
19     return x + y;
20 }
```

1.3 Primitive data types

In Java, primitive data types are predefined by the language and are named using reserved keywords. The eight primitive data types supported by the Java programming language are: **byte**, **short**, **int**,

long, float, double, boolean, char. The **String** class is a special case that is treated analogously to a primitive data type. Primitive type by themselves cannot be modified by reference (no pointers) and don't have class methods, i.e.

```

1 double x = 1.5;
2 x.round(); // Invalid, no class methods for primitive types
3 double y = x; // Valid, y equals 1.5
4 x = 2; // Valid, but y still equals 1.5: no reference for primitive types

```

The JRE contains class wrappers for each primitive type. These wrappers provide additional operations (casting, truncating, etc.) and allow primitives to be treated like objects (that is, they can be passed by reference). The naming convention for these classes is the name of the primitive type they reference with the first letter in uppercase (e.g. Double, Integer, Float, etc.).

1.4 Operators

Here are some examples showing the use and effect of common operators:

```

1 int x = 1;
2 int y = 1;
3 int z;
4
5 // Additive, multiplicative
6 z = x + y; // z is the sum of x and y
7 z = x - y; // z is the difference of x and y
8 z = y * x; // z is y times x
9 z = x / y; // z is x divided by y
10 z = x % y; // z is the remainder of x divided by y (modulus)
11
12 // Assignment
13 x += 2; // x is incremented by 2
14 x *= 2; // x is multiplied by 2
15 x /= 2; // x is divided by 2
16
17 // Postfix
18 x++; // x is incremented by 1 after "x" is executed
19 x--; // x is decremented by 1 after "x" is executed

```

Other operators include unary, shift, relational, bitwise, logical and ternary operators (see Oracle's Java Operators for a complete list along with the operator precedence).

String objects can be manipulated using the `+` sign, which acts as a concatenation operator:

```

1 String s = "Programmers " + "using C ";
2 s += " can't use this";
3 // s equals "Programmers using C can't use this"

```

1.5 Methods

Methods in Java have a signature similar to those in C. Methods are required to have a return type – **null** if nothing is returned, just like **void** in C –, a name, a comma-delimited parameter list in a parenthesis, and a body enclosed in brackets, e.g.

```

1 int sum(int x, int y) {
2     return x + y;

```

3| }

Optionally, you can specify a modifier such as **public** or **private**, depending on the scope of the method, and **static**. See Section 1.10 on Java classes for more on this. Other elements include an exception list (not covered in this tutorial) and generic types (also not covered).

1.6 The Main Method

Second most important part of your application, the **Main** method. This method is how the Java Virtual Machine knows that your application wants to be compiled. Without it, your code is just a complicated arrangement of spaghetti but without sauce. This method must appear within a class, but it can be any class. Here is an example of a *main()* method invoking the constructor for the application living in the class **MyApp**:

```
1 /**
2  * @param args
3  */
4 public static void main(String [] args) {
5     new MyApp();
6 }
```

More onto constructors and classes later...

1.7 Rounding

If you cast a double or a float directly to an integer, the decimal part of the number is simply omitted, no rounding operation is performed. e.g.

```
1 int x = (int) 1.7; // x equals 1
```

If you want “proper” rounding, then you can use the static *Math.round(number)* method, which returns the closest integer to the number, i.e. a **long** for a **double**, and an **int** for a **float**.

```
1 long x = Math.round(1.6) // x is a long rounding of the number
2 int y = Math.round(1.6f) // y is an int rounding of the number
3 int z = (int) Math.round(1.6) // z is a long rounding of the number, cast to an int
```

Other ways of rounding numbers include *Math rint(number)* and the *BigDecimal* class which has many rounding modes available.

1.8 Logic

Logical statements in Java have the following syntax. Note that brackets are only necessary when an **if** statement encloses more than one line of code.

```
1 double z = new Random().nextDouble(); // z is a random number between 0 and 1
2
3 if (z == 0) {
4     System.out.println("z equals 0 and we don't like this");
5     exit(1);
6 }
7 else if (z > 0 && z < 0.5)
8     System.out.println("z is nonzero and less than 0.5");
9 else if (z >= 0.5 && z < 1)
10    System.out.println("z is equal or larger than 0.5 and less than 1");
11 else if (z == 1) {
12     System.out.println("z equals 1 and we don't like this either");
13     exit(1);
14 }
15 // This should never happen
16 else {
17     System.out.println("wtf?");
18 }
```

Switch statements can be used when working with discretized values. The **break** keyword prevents the code from “leaking” from one case to another. The **default** keyword indicates the operation to be performed if the evaluation does not fall in any **case** block.

```

1 public String translateFrenchEnglishItalian(int x) {
2     String s;
3     switch (x) {
4         case 1:
5             s = "un, one, uno";
6             break;
7         case 2:
8             s = "deux, two, due";
9             break;
10        // cases 3-9...
11        case 10:
12            s = "dix, ten, dieci";
13            break;
14        default:
15            System.err.println("This translation engine only supports numbers up to 10.");
16            s = "invalid number";
17    }
18
19    return x + " is translated to [" + s + "];"
20 }

```

You can save a few lines when performing logical assignments by using an inline if-then-else statement:

```

1 double z = -1 + 2 * new Random().nextDouble(); // z is a number between -1 and 1
2 int x = z < 0 ? -1 : 1; // inline logic: x is the sign of z
3 int y = z < 0 ? -1 : z == 0 ? 0 : 1; // inline logic: y is the true signum of z
4 boolean b = y == 1; // b is true if z is strictly positive

```

1.9 Loops

A **for** loop is a block of code that is designed to be executed a number of times, until a certain condition is met, with an optional incremental operation. e.g.

```

1 // Incremental operation = i++
2 for (int i = 0; i < 10; i++) {
3     System.out.println("The current iteration is " + i);
4 }
5 // No incremental operation in loop declaration
6 for (int i = 0; i < 10; ) {
7     System.out.println("The current iteration is " + i);
8     i = i + 1;
9 }

```

A **while** loop is a block of code that is designed to be executed until a certain condition is met. e.g.

```

1 int i = 0;
2 while (i < 10) {
3     System.out.println("The current iteration is " + (i++));
4 }

```

A **do-while** loop is a loop that is designed to be executed once, and then until a certain condition is met. e.g.

```
1 do {  
2     hitLaptop();  
3 } while (isScreenFrozen());
```

1.10 Classes

A class is defined in a file having the name of the class and ending with the *.java* suffix. e.g.

```
1 // HumanInteractor.java  
2 public class HumanInteractor {  
3  
4     private final static int minIndex = 0;  
5  
6     public boolean isValid(int interactorIndex) {  
7         return interactorIndex >= minIndex;  
8     }  
9 }
```

Many objects of the same Class can be defined on the same line, e.g.

```
1 HumanInteractor human1 = new HumanInteractor(), human2 = new HumanInteractor();
```

The static keyword indicates that this field is shared by all objects implementing this class. The same thing can be done for classes (e.g. public static void main(String[] args).

A new class is constructed by calling the *new* keyword followed by the class name with parameters enclosed in brackets, e.g.

```
1 SomeClass object = new SomeClass("magic number", 42, 1.0f);
```

Anonymous classes can also be created on-the-fly if you don't need to keep track of them. In this example, an ActionListener object is being declared anonymously:

```
1 button.addActionListener(new ActionListener() {  
2     public void actionPerformed(ActionEvent e)  
3     {  
4         // do something.  
5     }  
6 });
```

Interface are classes that define a set of method signatures that a class implementing them must declare. e.g.

```
1 // Interactor.java  
2 public interface Interactor {  
3     public void attach(String component);  
4 }
```

Classes implementing the interface need to implement its methods:

```
1 // HumanInteractor.java  
2 public class HumanInteractor implements Interactor {  
3  
4     // This method has to exist in the HumanInteractor class  
5     // since it implements the Interactor interface which  
6     // declares it.
```



```
7  @Override
8  public void attach(String component) {
9      System.out.println("Attaching " + component);
10 }
11 }
```

1.11 JRE Convenience Classes

The Java JRE has many built-in classes that you will find useful for your implementation. You can trust these classes as they are well-maintained, updated anytime a new JRE comes out, and have been around for a long time. By using these classes, you won't need to reinvent the wheel whenever you need a new feature and will save a lot of time. Just make sure you read the documentation properly so you know exactly what to expect.

1.11.1 Lists

Java defines a **List** interface to represent any ordered collection. The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. Unlike sets, lists typically allow duplicate elements. The elements can be anything that is a Class (meaning no primitive types are allowed so in order to use them you will need their Class wrappers: Double, Integer, Float, etc.) and even the **null** element. You can traverse a List rapidly by using the **for** (*element : list*) syntax:

```
1 List<Vector3d> vectors = new LinkedList<Vector3d>();
2 points.add(new Vector3d(1, 2, 3); // element 0
3 points.add(new Vector3d(4, 5, 6); // element 1
4 points.add(new Vector3d();        // element 2
5 points.add(null);                 // element 3
6
7 // Traverse the list one element at a time.
8 // It will throw an exception if it reaches a null element and
9 // you try to call a method on it.
10 for (Vector3d v : vectors) {
11     v.normalize(); // throws an exception when it reaches the last element
12 }
```

Java comes with two standard and useful List subclasses: the **LinkedList** and the **ArrayList**.

LinkedList The LinkedList is a data structure consisting of a group of elements that form a sequence. Each element has a reference to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence. More precisely, Java uses a doubly-linked list. The running time for common operations is the following:

- **list.get(index)**: $O(n)$.
- **list.add(object)**: $O(1)$.
- **list.remove(index)**: $O(n)$. The constant factor is large compared to that for the ArrayList.

ArrayList The ArrayList is a dynamically-resizable array implementation of the List interface. Each ArrayList instance has a capacity that is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. You can increase the capacity of an ArrayList manually before adding elements using the *ensureCapacity(size)* method, reducing the amount of incremental reallocation. The running time for common operations is the following:

- **list.get(index):** $O(1)$.
- **list.add(object):** amortized $O(1)$, worst-case $O(n)$ if the array must be resized and copied.
- **list.remove(index):** $O(n)$. The constant factor is low compared to that for the LinkedList.

Collections This class allows to perform operations on Classes implementing the **List** interface. Useful operations include searching, sorting, shuffling, replacing, etc.

Arrays This class allows to perform operations on arrays containing primitive types, such as copying, searching, filling and sorting.

```
1 int[] array = new int[] { 4, 3, 2, 1 };
2 int[] sortedArray = Arrays.copyOf(array, 3); // Copies the first three elements.
3 sortedArray.sort();
```

1.12 Vecmath package

To complete the assignments in this course you will be working with matrices, points, and vectors. The Vecmath package provides useful classes for manipulating vectorial data, including 2D and 3D vectors and points, 3×3 , 4×4 matrices, axis angles, and quaternions. The Vecmath package is available as part of the (extensive) Java3D project, but instead of installing the full Java3D, you can download just the vecmath.jar locally. The source code (i.e., the javadoc) is in the jar but it might not be automatically attached. In order to attach the source code to the jar for the javadoc, first add the jar to your build path, then open up Reference Libraries in your project, right click on the jar to set properties, go to source code attachment, and tell it where to find the zip file.

1.12.1 Points and Vectors

Come in double or float flavour, and for two-, three-, and four-element tuples, e.g. Vector2d, Vector3f, Point2d, Vector4d. The components of 2-,3-,4-tuples can be respectively accessed directly using the public fields (x, y) , (x, y, z) , and (x, y, z, w) . Here is how you create them:

```
1 Point3d p = new Point3d(1, 2, 3);
2 Vector4d v = new Vector4d(); // Set v to (0, 0, 0, 0) by default
```

Here are some useful methods for these classes.

- Negate all components of this vector in place.

```
1 vector.negate();
```

- Scale all components of this vector by 2.

```
1 vector.scale(2);
```

- Sum up a point and a vector and place the result in the point.

```
1 point.add(vector);
```

- Sum up two vectors and place the result in the first.

```
1 vector1.add(vector2);
```

- Set the first vector as the sum of two others.

```
1 vector1.add(vector2, vector3);
```

- $\text{vector1} = 0.5 * \text{vector2} + \text{vector3d}$

```
1 vector1.scaleAdd(0.5, vector2, vector3);
```

- Compute the Euclidean distance between two points.

```
1 point1.distance(point2);
```

- Normalize a vector in place under the Euclidean distance so that its Euclidean norm is 1.

```
1 vector.normalize();
```

- Compute the length of a vector under the standard Euclidean distance.

```
1 vector.length();
```

Note: although programmatically they might seem identical, points and vectors are different. A point is a **location** (single point) and a vector is a **magnitude** and **direction** (an ordered pair of points). This distinction is the reason you cannot get the distance between two Vector2d or that you don't have a *scale()*, *length()* or *normalize()* method in the Point3d class.

1.12.2 Matrices

Come in double or float flavour, and for 3×3 and 4×4 matrices. The components of these matrices can be accessed directly using the public fields *m.ij*, e.g.

```
1 Matrix4d m = new Matrix4d();
2 m.00 = 1; // Set the first row, first column
3 m.01 = 2; // Set the firsts row, second column
4 m.33 = 16 // Set the fourth row, fourth column
```

Here are some useful methods for these classes.

- Compute the determinant of a matrix.

```
1 matrix.determinant();
```

- Sum up two matrices and place the result in the first.

```
1 matrix1.add(matrix2);
```

- Set the value of a matrix to a counter-clockwise rotation about the x axis.

```
1 matrix.rotX(Math.PI / 3);
```

- Transform a point using a matrix (dimensions must match).

```
1 matrix.transform(point);
```

- Retrieves the value at the specified row and column of this matrix.

```
1 matrix.getElement(0, 3);
```

Note: Extracting the rotation matrix from a 4x4 matrix using the *matrix4d.get()* method involves an SVD normalization and should be avoided since it is prone to numerical errors. If a 4x4 matrix is a rigid transformation matrix, you can use the *matrix4d.getRotationScale()* to extract the upper 3x3 matrix of the transformation matrix. This 3x3 matrix is in fact a combination of a rotation matrix and of a scaling matrix so don't forget to set its diagonal elements to 1.

1.12.3 Quaternions

Quaternions are a special number system first described by Hamilton in 1843. This number system has a convenient mathematical notation for representing rotations of objects in three dimensions. They are more numerically stable and may be more efficient than rotation matrices. In vecmath, quaternions are represented as a vector of their components, in double or float flavour. The components of a quaternion can be accessed directly using the public fields *x, y, z, w*, e.g.

```
1 Quat4d q = new Quat4d(); // q is (0, 0, 0, 0)
2 q.y = 1; // Set q to (0, 1, 0, 0)
```

2 Eclipse

This section will help you download and install Eclipse, and set up your Java workspace. It will also give you some tips on how to be more efficient while programming your assignments, so even if you have used Eclipse before, I recommend you at least skim over this section briefly.

You are strongly encouraged to use Eclipse as a development environment for this course. You are obviously free to develop in whatever IDE you prefer but we will only give technical support for Eclipse and your assignments will be graded using Eclipse.

2.1 History

Eclipse is a free and open source software development environment written mostly in Java. Although originally designed for Java developers, plugins and extensions now allow Eclipse to be also used to develop applications in languages including Ada, C, C++, COBOL, Fortran, Mathematica, Perl, PHP, Python, R and Ruby.

2.2 Verifying if Java is Installed

Eclipse requires Java to be installed in order to run properly. Nowadays, many operating systems come out of the box with a preinstalled Java **JDK** (Java Development Kit) and OS X is one of them. Almost all operating systems come with the Java **JRE** (Java Runtime Environment) installed so you shouldn't have to worry about this one. The JDK and JRE respectively consist of a compiler to compile your Java code to bytecode and a virtual machine to run your Java programs. Note that you may have other Java compilers (e.g. Microsoft or GNU Java) installed on your machine, but **Sun / Oracle's** Java will be used for this course.

Here is a simple test to determine if the Java JDK is already installed on your machine. If you are using OS X, Unix or Linux, type **which java** in a console and if the output shows a location (e.g. `/usr/bin/java`) then you should have a JDK installed. If you are using windows, you can search for **javac.exe**, which is the Java compiler. You can dig a little deeper to determine what version of the Java JDK you have. For this course, we will be using a Java JDK 1.6 compliance level so make sure your version is at least 1.6 (as this tutorial is given, the most recent version is 1.7). If you have an older version, you might not be able to compile certain parts of the base code we provide you for the assignments.

Quick solution If the previous test fails or if you are unsure about all this and just want to start your assignment as soon as possible, you can just go ahead and download the latest Java SE JDK – just be careful not to select the JRE – for your machine at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

2.3 Installation

You can download Eclipse at <http://www.eclipse.org/downloads/>. There are many packages available for download. They essentially all share the same basic codebase, but have different plugins preinstalled, depending on what your needs are. You can eventually decide to use something else but for this course, download the **Eclipse IDE for Java Developers**. Make sure you select the appropriate computer architecture for your machine (32 or 64 Bit) and operating system.

2.4 Configuring JREs and JDK Compiler Compliance

This next step only has to be performed once, so you should go through it and make sure everything is fine. We've all had headaches in the past because of a mismatch in the JRE version and the compiler compliance level. In order to avoid compatibility issues, you should tell Eclipse to use a JRE version of 1.6 (or more) and a JDK compliance level of 1.6 (or more). In order to do that, you need to change some Eclipse preferences.

In OS X, click **Eclipse** → **Preferences** and in Windows, click **Window** → **Preferences**. On the left, select **Java** → **Installed JREs**. On the right, check the default JVM to use, e.g. JVM 1.6 (there can be multiple versions installed with the same number, just pick one). If no JVM 1.6+ exists, then look at Section 2.2 on how to install Oracle's Java JDK.

On the left, select **Java** → **Compiler**. On the right, under **JDK Compliance**, set the **Compiler compliance level:** to 1.6+. Again, if you can't do that, then look at Section 2.2.

2.5 Workspace Management

Eclipse uses a **workspace** to organize your projects. A workspace is simply a directory in which Eclipse places some hidden configuration files. When you start Eclipse for the first time, it will automatically create a default workspace at some location (e.g. in your home directory).

You can switch to or create a new workspace by going to **File** → **Switch Workspace** → **Other...**, and by selecting any folder you want for your workspace. For example, you could use a unique workspace for this course by creating a folder named “workspaceComp557” in your home directory, and then telling eclipse to switch to that workspace. Whenever you launch Eclipse, it loads the workspace that was used during its previous utilization.

2.6 Perspectives, Views and Packages

Eclipse has a system of “perspectives” and “views” for managing its layout and the elements that are displayed in its interface. In the menu, under **Window** → **Show View**, you have a list of available displayable elements. For example, the Package Explorer lets you manage your Java code and projects efficiently by clustering them into packages. We will be using this feature in this course so you will get familiar with it. A class belongs to a package by using the reserved **package** keyword at the top of a Java class file, e.g.

```
1 // Dots are used to separate subpackages
2 // In this case, ''edu'' is the top-level package,
3 // containing the ''mcgill'' package
4 // which contains the ''interaction'' package:
5 package edu.mcgill.interaction;
6
7 import javax.media.opengl.GL;
8
9 public class HumanInteractor implements Interactor {
10     ...
11 }
```

A **perspective** is a predefined set of views along with a layout that you can save/load. You can load and move the different views around depending on your preference and then save your own perspective in **Window** → **Save Perspective As...**

2.7 Creating a Java Project

Once you are in the appropriate workspace, you can create a new java project. Click **File** → **New** → **Java Project**. Give your project a name, such as “assignment0”. Now look under **JRE**. By default, Eclipse will use your most recent JVM version. Make sure that version is at least **1.6**. If it’s not the case, then select **Use a project specific JRE:** and pick an appropriate JVM version. If no JVM 1.6+ exists, then look at Section 2.2 on how to install Oracle’s Java JDK and Section 2.4 on how to configure Eclipse defaults with the appropriate JVM.

In the same window, under **Project layout**, select “Create separate folders for sources and class files”, which will make the folder in which your code is stored a lot cleaner.

In the same window, under **Working sets**, unselect “Add project to working sets” unless you intend to that feature (it actually is a useful feature so read about it if you can).

Now click Finish. Et voilà, you're done. Once the project is created, it should appear under your Package Explorer. The project should contain a **src** folder, a **Referenced Libraries** folder and a **JRE System Library** folder.

2.8 Creating a Folder, Package, Class, Interface, Enum or Others

From the package explorer, if you right click in the appropriate folder or package, **New** → lets you create a new folder, package, subpackage, Java class, Java interface, or others. A subpackage is created by using the full path of its parent (e.g. edu.mcgill) and then adding a dot followed by the package name (e.g. "edu.mcgill.interaction" for a new subpackage interaction located in the "edu.mcgill" package).

2.9 Must-Read: Tips and Tricks

Here are some tips on how to better use Eclipse when implementing and browsing your code.

Comment / Uncomment You can comment and uncomment multiple lines of code at the same time by selecting them with your mouse and pressing the keyboard shortcut **Command**+**/** (forward slash) in OS X or **Ctrl**+**/** in Linux or Windows.

Navigation History The editor keeps a navigation history so you can easily go backward and forward to a previous or next source file you've edited. In OS X, press **Alt**+**Command**+**(Left or Right)** and in Linux or Windows, **Alt**+**(Left or Right)** to navigate in that manner. You can also use the yellow forward and back buttons in the toolbar.

Javadoc Hover If you let your mouse hover over an identifier, variable or method, a window will pop up telling you what it is, and providing you with the java documentation about this element (javadoc).

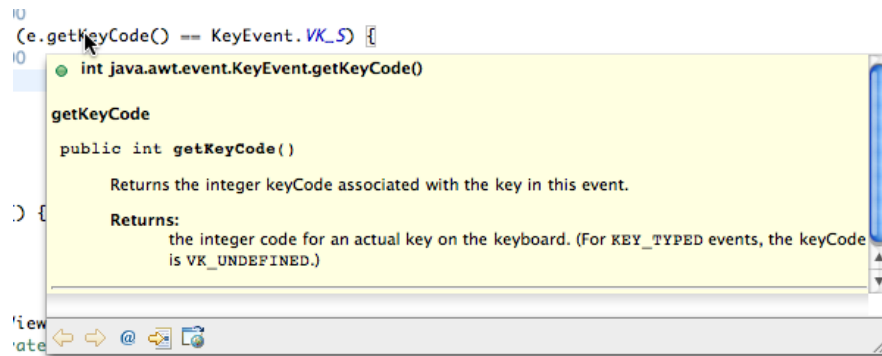


Figure 1: Mouse hover brings up the javadoc

Jump to Definition If you hold the **Command** key (OS X) or **Ctrl** key (Windows, Linux) and **left-click** on an identifier, it will take you to its definition (source code). An identifier can be an object, a field, a class, a method, pretty much anything. Combine this with the navigation history trick and you will be exploring your code much faster. Note that if you do this on a method that is defined in a *jar*, there might be no definition available if the jar does not contain the source code (or was not attached to it).

```
@Override
public void keyPressed(KeyEvent e) {
    controlDown = e.isControlDown();
    shiftDown = e.isShiftDown();
}
```

Figure 2: Command-click (OS X) or Ctrl-click (Linux, Windows) to jump to definition.

Call Hierarchy If you ever wonder what classes are calling a certain method, you can right-click on that method, then select **Open Call Hierarchy**. A new view will be shown with all members that call this method.

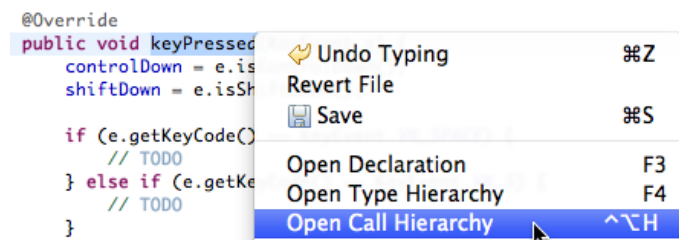


Figure 3: Opening the call hierarchy for a method.

And Many More... For more ideas on how to improve your browsing, coding experience and productivity, have a look at this official Eclipse document (Eclipse documentation - Current Release, under Workbench User Guide: Tips and tricks).

2.10 Code refactoring

Code refactoring is a technique for restructuring an existing body of code without changing its external behavior. For instance, if you want to change the name of a class, you expect the resulting “rename” to propagate to all classes that are concerned with that change, without affecting the way the code executes. Typically, refactoring is done by applying series of small changes to the source code that do not modify its functional requirements. Advantages include improved readability, maintainability, and extensibility of the source code. You should try to use refactoring as often as possible, whenever necessary. As a piece of software grows, many classes will naturally need to be renamed and moved around.

As an example, if you right click on a method in an interface class, then select **Refactor** →, here is a snapshot of what the refactoring menu looks like:

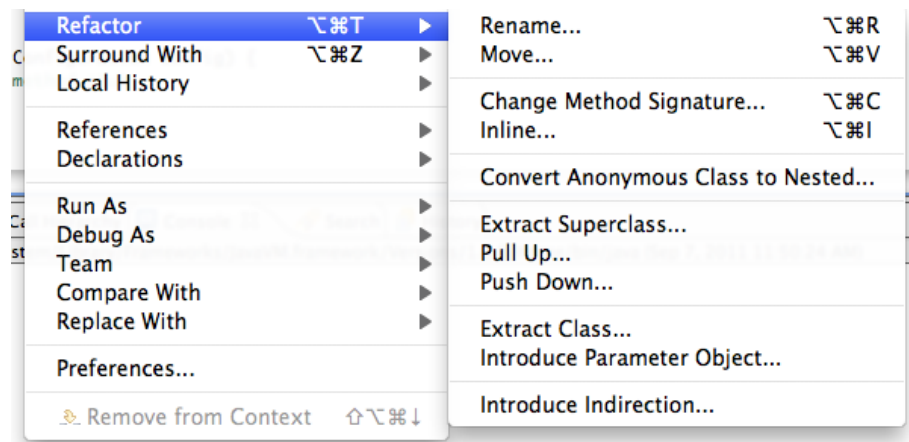


Figure 4: Eclipse refactoring menu for an interface method in OS X

The refactoring menu can be brought up by right-clicking pretty much any element (object, class, method, etc.) in the source code editor. Some refactoring actions are more useful than others and can be applied on many kinds of elements. Here is a short list of useful refactoring actions.

- **Rename...** lets you rename this element. **alt+command+R** in OS X and **alt+shift+R** in Linux or Windows.
- **Move...** lets you move this element to a new location (package or class). **alt+command+V** in OS X and **(alt)+shift+(V)** in Linux or Windows.
- **Change Method Signature...** does just that, for instance changing the return type of a method or changing its parameter list. **alt+command+C** in OS X and **alt+shift+C** in Linux or Windows.
- **Push Down...** moves a set of methods and fields from a class to its subclasses.
- **Pull Up...** moves a field or method to a superclass of its declaring class or (in the case of methods) declares the method as abstract in the superclass.

For other more specific commands, have a look at the documentation (Eclipse documentation - Current Release, under Java development user guide: Refactoring).

2.11 Debugging and Hot Code Replace

Debugger The Eclipse Java debugger features many standard debugging functionalities, including the ability to set breakpoints, to perform step into/over/return actions, to inspect objects and values, to evaluate expressions, and to suspend/resume/terminate threads. To debug your application, select **Run** → **Debug As** → **Java Application** or click the **debug** button:



Figure 5: Launch your application in debug mode by clicking the debug button.

Hot code replace HCR is a debugging technique that allows you to make changes to your code while debugging and see the effect in real-time. The Eclipse debugger can replace old class files in the Virtual Machine while it is running. No restart is required, hence the reference to “hot”.

From the Eclipse FAQ: “[...] HCR has been specifically added as a standard technique to Java to facilitate experimental development and to foster iterative trial-and-error coding. HCR only works when the class signature does not change; you cannot remove or add fields to existing classes, for instance. However, HCR can be used to change the body of a method. HCR is reliably implemented only on 1.4.1 VMs and later [...]”.

You need to be running your application in **debug mode** in order for HCR to work. If HCR still does not work, make sure you have **automatic building** turned on by checking **Project** → **Build Automatically**.

2.12 JVM Heap Space and Out of Memory Exceptions

The exception As you start to build programs that become more memory hungry (adding a large number of heavy objects to a list, for instance), you might end up getting an exception of the type:

```
1 Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

The operating system automatically allocates a certain amount of working memory (heap space) to the Java Virtual Machine when your program is started. However, the default size of the heap is typically **128MB**. If your program exceeds that memory limit, the JVM will throw the exception message seen above.

The fix You can increase the size of the JVM heap space by adding the appropriate virtual machine arguments to your program’s run configuration. Click **Run** → **Run configurations...** then select the **Arguments** tab. Under **VM arguments**, specify the minimum and maximum heap size. **-Xms** sets the minimum heap size (in megabytes, don’t forget the **m** after the number) the JVM should use for your program and **-Xmx** sets the maximum.. For example, if you want a minimum of 128MB and maximum of 512MB, use:

```
1 -Xms128m
2 -Xmx512m
```

Likewise, you can increase the heap size for your debugging sessions by going in **Debug** → **Debug configurations...** and repeating the same process as for your run configurations.

3 Swing and MINTtools

During this course, you will be using the **Mintools** Java framework for designing user interfaces (UI), and displaying OpenGL code. The Mintools framework is a combination of convenience classes

for rapidly constructing and manipulating a Java Swing UI, including an OpenGL canvas. It was developed in the Computer Graphics Lab at McGill.

3.1 Master of the UI: Creating The EasyViewer

In order to create your UI, you will first need to create an instance of the **EasyViewer** class. The EasyViewer class is a convenience class that allows you to create, manage and expand the main UI of your application easily. It also automatically creates an OpenGL window. This tutorial will not discuss any OpenGL-related topics; attend next tutorial for that instead.

The EasyViewer constructor requires

- a title for your UI window frame,
- a class (your application) implementing the **SceneGraphNode** interface for handling the creation of the UI and the OpenGL calls,
- a dimension for the OpenGL canvas
- and a dimension for the frame of your UI.

You can create an instance of the EasyViewer class in your **main** by directly creating an instance of your application class:

```

1 public static void main(String[] args) {
2     new EasyViewer("Some app title", new MyApp(), new Dimension(640,480), new
        Dimension(320,480) );
3 }
```

or better, you can create it inside your class constructor and then add a reference to your class from it. This allows you to manipulate the EasyViewer directly in your code:

```

1 public class MyApp implements SceneGraphNode {
2
3     public static void main(String[] args) {
4         new MyApp();
5     }
6
7     // Application constructor
8     public MyApp() {
9         // Create the EasyViewer
10        String title = "Some app title";
11        SceneGraphNode app = this;
12        Dimension glSize = new Dimension(640, 480);
13        Dimension frameSize = new Dimension(320, 480);
14        EasyViewer ev = new EasyViewer(title, app, glSize, frameSize );
15
16        // We now have a reference to the EasyViewer
17        ev.doSomething();
18    }
19
20    @Override
21    public void init( GLAutoDrawable drawable ) {
22        // Init your OpenGL code here
23    }
24 }
```

```

25  @Override
26  public void display( GLAutoDrawable drawable ) {
27      // Draw your OpenGL scene here
28  }
29
30  @Override
31  public JPanel getControls() {
32      // Return your UI controls here
33  }
34
35 }

```

3.2 Building your UI

3.2.1 The getControls() method

During runtime, and only once, the EasyViewer will automatically call your application's *getControls()* method. In this method you can add elements to your UI:

```

1  @Override
2  public JPanel getControls() {
3      // Create the panel that will contain our UI elements.
4      VerticalFlowPanel vfp = new VerticalFlowPanel();
5
6      // Add a label to the panel.
7      vfp.add( new JLabel( "hello f00bar" ) );
8
9      // Return the panel containing our UI
10     return vfp.getPanel();
11 }

```

You have a direct access to the frame that contains your UI through the EasyViewer's **controlFrame** field. This frame also allows you to add tabs to your application, for instance:

```

1  JPanel panel = new JPanel();
2  easyViewer.controlFrame.add( "New Panel", panel );

```

3.2.2 CollapsiblePanel

A **CollapsiblePanel** is used for showing and hiding a panel that contains elements (see Figure 6). You create it by adding the panel directly to it:

```

1  @Override
2  public JPanel getControls() {
3      VerticalFlowPanel vfp = new VerticalFlowPanel();
4      CollapsiblePanel cp = new CollapsiblePanel( vfp.getPanel() );
5      return cp;
6  }

```

3.3 Parameters

Mintools contains a set of useful parameters for manipulating your data using a user interface. They act as wrappers around Java primitive types including a **BooleanParameter**, a **DoubleParam-**

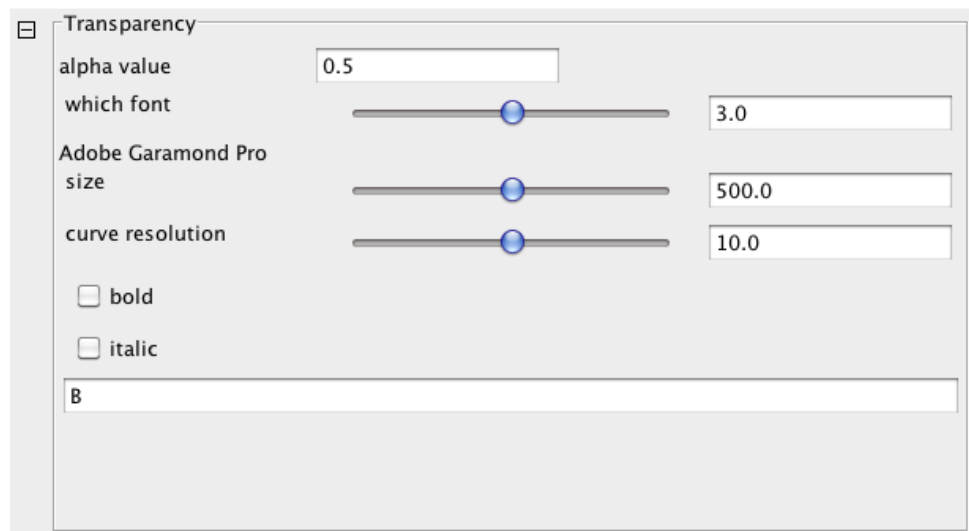


Figure 6: A collapsible panel containing diverse UI elements. The collapsible button is shown as a minus sign in a square at the top left of the frame.

eter, and an `IntParameter` class. As their name suggests, these classes are respectively used for representing boolean, double, and integer values.

3.3.1 Constructors

The `BooleanParameter`, `DoubleParameter`, and `IntParameter` all have a **value**, either logical (true or false) or numerical (real or integer), and a **default value**. The `DoubleParameter` and `IntParameter` also have a **minValue**, and a **maxValue**.

Here is how you initialize them:

```
1 BooleanParameter bparam = new BooleanParameter( "some text", defaultValue );
2 DoubleParameter dparam = new DoubleParameter( "some text", defaultValue, min, max );
3 IntParameter iparam = new IntParameter( "some text", defaultValue, min, max );
```

3.3.2 Controls

You can add UI controls for your parameters by calling their `getControls()` method:

```
1 @Override
2 public JPanel getControls() {
3     VerticalFlowPanel vfp = new VerticalFlowPanel();
4     BooleanParameter bparam = new BooleanParameter( "boolean", true );
5     DoubleParameter dparam = new DoubleParameter( "double", 0, 0, 1 );
6     vfp.add( bparam.getControls() );
7     vfp.add( dparam.getSliderControls( false ) );
8     return vfp.getPanel();
9 }
```

Figure 7 shows the controls for a BooleanParameter and Figure 8 shows controls for a DoubleParameter.

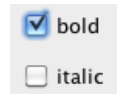


Figure 7: BooleanParameter UI: name (JLabel) and true / false value (JCheckBox).



Figure 8: Numerical parameter UI: name (JLabel), slider (JSlider), and value (JTextField).

3.3.3 Listeners

You need to set up a **ParameterListener** if you want to be able to react to the changes applied to the parameters in the UI. For instance, if you want to print to the console the value of a DoubleParameter whenever it is changed, you would create an instance of a ParameterListener that does just that and then add it to the parameter.

Note that the ParameterListener class is an interface so you must create an instance of the interface in order to use it. Generally, this is done by creating an **anonymous inner class** using the following syntax:

```

1 // Create some DoubleParameter object
2 DoubleParameter dparam = new DoubleParameter( "some text", defaultValue, min, max );
3
4 // Create an anonymous class implementing the ParameterListener interface
5 ParameterListener listener = new ParameterListener() {
6     @Override
7     public void parameterChanged( Parameter parameter ) {
8         System.out.println( dparam.getValue() );
9     }
10 };
11
12 // Add the listener to the parameter
13 dparam.addParameterListener( listener );

```

Note that you can add a ParameterListener to multiple Parameter objects. You can also parse the name of the Parameter that triggered a change:

```

1 ParameterListener listener = new ParameterListener() {
2     @Override
3     public void parameterChanged( Parameter parameter ) {
4         System.out.println( parameter.getName() );
5     }
6 };
7 booleanParam1.addParameterListener( listener );
8 booleanParam2.addParameterListener( listener );
9 intParam1.addParameterListener( listener );
10 ...

```

3.4 Keyboard and Mouse Input

The EasyViewer naturally supports keyboard and mouse input. By having your application implement the **Interactor** interface and calling the *addInteractor()* method of the EasyViewer, you can set up keyboard and mouse handling:

```
1 // In your App constructor
2 easyViewer.addInteractor( this );
```

In order to implement the **Interactor** interface, you will need to define the *attach()* method, in which you can decide what kind of input you are interested in. For instance, if you want to respond only to key presses:

```
1 @Override
2 public void attach(Component component) {
3     component.addKeyListener(new KeyAdapter() {
4         @Override
5         public void keyPressed(KeyEvent e) {
6             if (e.getKeyCode() == KeyEvent.VK_SPACE) {
7                 // Do something
8             }
9             else if (e.getKeyCode() == KeyEvent.VK_Q) {
10                System.exit( 0 );
11            }
12        }
13    });
14 }
```

The advantage of using the **KeyAdapter** is that you don't need to implement the other keyboard handling methods if you don't need them, such as *keyTyped()* and *keyReleased()*.

The same applies for mouse handling, although you need to treat mouse motion separately from other mouse events (such as clicking). The *addMouseListener()* allows you to handle all mouse events except motion, including *mouseClicked()*, *mouseEntered()*, *mouseExited()*, *mousePressed()*, and *mouseReleased()*. If you only need to know when and where the user actually clicked (meaning click and release) something, use the *mousePressed()* method:

```
1 @Override
2 public void attach(Component component) {
3     component.addMouseListener(new MouseAdapter() {
4         @Override
5         public void mousePressed(MouseEvent e) {
6             // You can also get the mouse position
7             Point pos = e.getPoint();
8
9             if (e.getButton() == MouseEvent.BUTTON1) {
10                // Left click
11            }
12            else if (e.getButton() == MouseEvent.BUTTON2) {
13                // Middle click
14            }
15            else if (e.getButton() == MouseEvent.BUTTON3) {
16                // Right click
17            }
18        }
19    });
20 }
```

If you need mouse motion and position information, use the **MouseMotionListener**:

```
1 @Override
2 public void attach(Component component) {
3     component.addMouseListener(new MouseMotionAdapter() {
4         @Override
5         public void mouseMoved(MouseEvent e) {
6             // Get the new mouse position
7             Point pos = e.getPoint();
8         }
9     });
10 }
```

You can also determine if special keys were held while a mouse event occurred, by calling the *isAltDown()*, *isShiftDown()*, or *isControlDown()* method on the `MouseEvent`.