1 Purpose

This document assumes that you're a good programmer but you don't know much Java. So you already know all about loops and classes, you're kinda comfortable with threading, and you know basic Java syntax (what you might get from an introductory Comp Sci course that uses Java), but you don't know some of the fancier aspects of Java (packaging, syntax of anonymous/nested classes, the Swing library, etc.).

That's where I was when I got onto this project, so this is pretty much supposed to be the guide to Java I wish I'd had from the start.

2 Compiling and running Java programs

I have this whole rant about Java IDEs, but I'll spare you. The punch line is that I hate using them to compile/debug/run my code, so I do everything from the command line using `javac` and `java`. If you want to use an IDE, I can't help you, but if you're fine with Bash, I can describe how to compile/debug/run a program.

Compiling is simple enough: just `cd` into the top level of the directory structure whose Java files you want to compile and issue this command:

```
javac ‘find . -name ’*.java’‘
```

or if your filenames have spaces in them (shame on you),

```
find . -name ’*.java’ -print0 | xargs -0 javac
```

Either of these commands will compile all the .java files in your current directory. If you're going to use `jdb` to debug your program, you'll want to use the `-g` option for `javac`.

To run, you just say `java ClassName` where `ClassName.class` is the class file that has the main method in it. In this case, the fully qualified name of the main class is `MMT.Main`, so you'll want to be in the directory that contains the `MMT` directory and say `java MMT/Main`.

To debug, do exactly the same thing, except use `jdb` instead of `java`. And you should probably have compiled the files using the `-g` option.

3 Packages

At the top of all the files in this project, you'll see the line

```
package MMT.blah.whatever;
```

where the file in question lives in the directory `MMT/blah/whatever`.

I'm gonna be honest here: I really don't understand packages. All I know is that for some reason, the package name has to mirror the directory structure up to some common top point. I'm sorry I don't know more.

4 Syntax

4.1 this as a function

`this` can be used as a function rather than an object reference. It just runs the constructor with whatever arguments you pass it. Like this!

```
public class Thing {
private int x;
    public Thing(int x) {
        this.x = x;
    }
    public Thing() {
        // initializes with value 0
        this(0);
    }
}
```

4.2 Anonymous classes

(used in `MMT/gui/ActuatorPanel.java`, `MMT/gui/TrackerPanel.java`)

Anonymous classes are how Java deals with callbacks (functions passed as arguments to other functions). For example, you'd like to run some code in the background without holding up the other parts of your program, like this:

```
function timeConsumingTask() returns void {
    doHardStuff();
}
doInBackground(timeConsumingTask);
```

But the Java syntax doesn't allow that. Instead, Java uses "anonymous classes," short-lived classes that are instantiated without ever being given a name. They look like this:

```
Thingamajig x = new Thingamajig(<initialization arguments>) {
    <subclass text>
};
```

That bit of code does what you'd expect from this bit of pseudo-code:

```
class TempClass extends Thingamajig {
    <subclass text>
}
Thingamajig x = new TempClass(<initialization arguments>);
```

We can use an anonymous class to run the `timeConsumingTask` in the background:

```
// In file BackgroundWorker.java
public abstract class BackgroundWorker {
    public abstract void run();
    public void doInBackground() {
        // executes this.run() in a background thread
    }
}

// In other file:
BackgroundWorker w = new BackgroundWorker() {
    public void run() {timeConsumingTask();}
};
w.doInBackground();
```

4.3 Local variables

Often, you'll want to embed some local variable into your anonymous classes. It turns out you can do this by simply referencing the variables inside the anonymous class, *as long as they are final.* For example, you might to write a function that takes two numbers, waits a second in the background, then prints their sum to the screen:

```
public void waitPrintSum(final int x, final int y) {
    BackgroundWorker w = new BackgroundWorker() {
        public void run() {
            Thread.sleep(1000);
            System.out.println(x + y);
        }
    }
    w.doInBackground();
}
```

This code will work! However, if `x` and `y` were not declared to be `final`, this would not compile.

5 Nested classes

Sometimes, two classes have such a close relation that you want one of them to be a part of the other. For example, the `Actuator` class has a nested class `Actuator.Status` that represents a status message received from the actuator, and the `HistoryPanel` class has a class `HistoryPanel.Updater` representing a task that updates the panel.

Nested classes behave in one of two ways, depending on whether the inside class is `static` or not. The way you should think about the difference is that

- when the inner class is not declared `static`, each instance of the inner class is associated with an instance of the outer class. The `HistoryPanel.Updater` class is not `static` because it needs to know which panel it's updating – each instance of `Updater` is fundamentally tied to a `HistoryPanel`.

- When the inner class is `static`, the inner class is associated with the outer class, but there's no relation between instances of one and instances of the other. The `Actuator.Status` class is `static` because the `Status` doesn't really care which `Actuator` (if any) generated it – it's just a container for some information that could've come from anywhere.

5.1 Nonstatic nested classes

If the inside class is not declared static, like this:

```
public class HistoryPanel {
    public class Updater {
        ...
    }
}
```

then each instance of `HistoryPanel.Updater` is inherently associated with an instance of `HistoryPanel`, and has access to the private variables of the enclosing `HistoryPanel` object. When you instantiate the `Updater`, you must specify which `HistoryPanel` it's associated with by modifying the `new` keyword slightly, like this:

```
HistoryPanel history = new HistoryPanel();
Updater updater = history.new Updater();
```

Here's a short example of a nonstatic nested class:

```
public class OuterClass {
    private int x;
    private OuterClass(int x) {this.x = x;}
    public class InnerClass {
        public void printX() {System.out.println(x);}
    }
    public static void main(String[] args) {
        OuterClass outer = new OuterClass(4);
        InnerClass inner = t.new InnerClass();
        inner. printX();
    }
}
```

When compiled and run, this program will just print "4".

5.2 Static nested classes

If the inside class is declared static, the fact that it's nested instead of living in its own file is almost just an organizational convenience. You instantiate the inner class using the normal `new` keyword and it acts just like a normal class, with the single exception that it has access to the static variables of the outer class. Here's what the previous example might look like if `InnerClass` had been declared static:

```
public class OuterClass {
    private static int staticVar = 4;
    private int instanceVar;
    public static class InnerClass {
        public void printStatic() {
            System.out.println(staticVar);
        }
    }
    public static void main(String[] args) {
        InnerClass inner = new InnerClass();
        inner.printStatic();
    }
}
```

Again, this just prints "4".

6 Swing

Good lord.