

**NEAR EAST UNIVERSITY**

**Faculty of Engineering**

**Department of Computer Engineering**

**EGG CATCHER GAME**

**Graduation Project**

**COM – 400**

**Student: Burak AKIN 20081375**

**Supervisor: Assist.Prof.Dr.Besime ERIN**

**Nicosia - 2014**

**NEAR EAST UNIVERSITY**

**Faculty of Engineering**

**Department of Computer Engineering**

**EGG CATCHER GAME**

**Graduation Project**

**COM – 400**

**Student: Burak AKIN 20081375**

**Supervisor: Assist.Prof.Dr.Besime ERIN**

**Nicosia - 2014**

## **ACKNOWLEDGEMENTS**

First, I would like to thank to supervisor of the project, Assist.Prof.Dr.Besime ERIN, for her invaluable advice and belief in my work and myself over the course of this project. Her guidance and encouragement helped me so much.

Second, I would like to express my gratitude to Near East University for providing magnificent environment and facilities throughout my education.

Third, I thank my family for their support and encouragement throughout my education and also during the preperation of this project.

## **ABSTRACT**

People regardless of their age or job have always been interested in playing games. Games helps people not only have nice time but also depending on the game it helps in so many other ways.

Computer games born out of our for lack of a better word “need” to play games and it gave us the ability to share our imagination with other people as accurately as we can.

When it comes to computer games there is no one single true approach regarding how to make a game or how should the game act. There are those games which are rely havily on being realistic and there are those that concern with action and reaction more then anything else.

The aim of this project is to produce a straight forward, easy to play computer game that is engaging and has a very low hardware requirements so it can be played on practically any computer by anyone.

## TABLE OF CONTENTS

<b>ACKNOWLEDGEMENTS</b>	<b>i</b>
<b>ABSTRACT</b>	<b>ii</b>
<b>TABLE OF CONTENTS</b>	<b>iii</b>
<b>INTRODUCTION</b>	<b>1</b>
<b>JAVA</b>	<b>2</b>
1.1 History	2
1.1.1 Principles	3
1.1.2 Versions	3
1.2 Java Platform	3
1.3 Implementations	4
1.4 Performance	5
1.5 Automatic Memory Management	5
1.6 Syntax	6
1.7 Hello World!	8
1.8 Applet	10
1.9 Servlet	12
1.10 JavaServer Pages	13
1.11 Swing Applications	13
1.12 Generics	15
1.13 Criticism	15
1.14 Use by External Companies	15
1.14.1 Google	15
1.14.2 Gaikai	16
1.15 Documentation	16
<b>LIGHTWEIGHT JAVA GAME LIBRARY</b>	<b>17</b>
2.1 Introduction	17
2.2 Goals of LWJGL	17
2.2.1 Speed	17
2.2.2 Ubiquity	18
2.2.3 Simplicity	18
2.2.4 Smallness	19

2.2.5 Security	19
2.2.6 Robustness	19
2.2.7 Minimalism	20
<b>SLICK2D ENGINE</b>	<b>21</b>
3.1 Introduction	21
3.2 Game Containers	21
3.2.1 Basic Functionality	21
3.2.2 Application Game Container	22
3.2.3 Applet Game Container	23
3.3 State Based Games	23
3.4 Setting up Slick2D with Eclipse	24
3.4.1 Introduction	24
3.4.2 Downloading and Extracting Slick2D and LWJGL	24
3.4.3 Setting Up Slick2D and LWJGL in Eclipse	25
3.4.4 Setting Up a Project to use LWJGL in Eclipse	25
<b>MARTE ENGINE</b>	<b>27</b>
4.1 Introduction	27
4.2 Features	27
4.2.1 Entity and World	27
4.2.2 Render and Update	28
4.2.3 ResourceManager	28
4.2.4 Basic Collision	29
<b>ECLIPSE</b>	<b>31</b>
5.1 Overview	31
5.2 History	31
5.3 Licensing	32
5.4 Name	32
5.5 Architecture	32
5.6 Rich Client Platform	33
5.7 Server Platform	34
5.8 Web Tools Platform	34
5.9 Modeling Platform	34
5.10 Model Transformation	35

5.11 Model Development Tools	35
5.12 Extensions	35
<b>DESCRIPTION OF THE APPLICATION</b>	<b>36</b>
6.1 Introduction	36
6.2 CONCEPT	37
6.3 Project Setup	38
6.4 Classes	39
6.4.1 EggCatcherGame Class	40
6.4.2 Menu Class	40
6.4.3 GameWorld Class	42
6.4.4 GameOver Class	43
6.4.5 Background Class	44
6.4.6 Chicken Class	45
6.4.7 Egg Class	45
6.4.8 Broken Class	46
6.4.9 Ground Class	46
6.4.10 Player Class	47
<b>CONCLUSION</b>	<b>48</b>
<b>REFERENCES</b>	<b>49</b>
<b>APPENDICES</b>	<b>50</b>

## **INTRODUCTION**

This project describes development of a game which is named Egg Catcher using LWJGL, Slick 2D Engine and Marte Engine on top of Java programming language.

It is my belief that this project will shed light of usefulness of tools like LWJGL, Slick 2D Engine and Marte Engine which are all open source and are developed by collaboration of imaginative and able programmers all around the world.

Game itself developed to provide nice time to people who like to spend their free time playing enjoyable and fun games and also because of its game play, Egg Catcher can be seen as an educational tool to help those who are not accustomed to computers and by extension don't accustomed to operating mouse.

The project consist of introduction, 6 chapters and conclusion.

Chapter One describes the Java Programming Language.

Chapter Two describes the LWJGL.

Chapter Three describes the Slick 2D Engine.

Chapter Four describes the Marte Engine.

Chapter Five describes the Eclipse text editor.

Chapter Six describes how this game has developed.



# JAVA

## 1.1 History

Java is a general-purpose, concurrent, class-based, object-oriented computer programming language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java applications are typically compiled to bytecode (class file) that can run on any Java virtual machine (JVM) regardless of computer architecture. Java is, as of 2012, one of the most popular programming languages in use, particularly for client-server web applications, with a reported 10 million users. Java was originally developed by James Gosling at Sun Microsystems (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

The original and reference implementation Java compilers, virtual machines, and class libraries were developed by Sun from 1991 and first released in 1995. As of May 2007, in compliance with the specifications of the Java Community Process, Sun relicensed most of its Java technologies under the GNU General Public License. Others have also developed alternative implementations of these Sun technologies, such as the GNU Compiler for Java and GNU Classpath.

### **1.1.1 Principles**

There were five primary goals in the creation of the Java language.

- It should be "simple, object-oriented and familiar"
- It should be "robust and secure"
- It should be "architecture-neutral and portable"
- It should execute with "high performance"
- It should be "interpreted, threaded, and dynamic"

### **1.1.2 Versions**

Major release versions of Java, along with their release dates:

- JDK 1.0 (January 21, 1996)
- JDK 1.1 (February 19, 1997)
- J2SE 1.2 (December 8, 1998)
- J2SE 1.3 (May 8, 2000)
- J2SE 1.4 (February 6, 2002)
- J2SE 5.0 (September 30, 2004)
- Java SE 6 (December 11, 2006)
- Java SE 7 (July 28, 2011)
- Java SE 8 (March 18, 2014)

## **1.2 Java Platform**

One characteristic of Java is portability, which means that computer programs written in the Java language must run similarly on any hardware/operating-system platform. This is achieved by compiling the Java language code to an intermediate representation called Java bytecode, instead of directly to platform-specific machine code. Java bytecode instructions are analogous to machine code, but they are intended to be interpreted by a virtual machine (VM) written specifically for the host hardware. End-users commonly use a Java Runtime Environment (JRE) installed on their own machine for standalone Java applications, or in a Web browser for Java applets.

Standardized libraries provide a generic way to access host-specific features such as graphics, threading, and networking.

A major benefit of using bytecode is porting. However, the overhead of interpretation means that interpreted programs almost always run more slowly than programs compiled to native executables would. Just-in-Time (JIT) compilers were introduced from an early stage that compile bytecodes to machine code during runtime.

### **1.3 Implementations**

Oracle Corporation is the current owner of the official implementation of the Java SE platform, following their acquisition of Sun Microsystems on January 27, 2010. This implementation is based on the original implementation of Java by Sun. The Oracle implementation is available for Mac OS X, Windows and Solaris. Because Java lacks any formal standardization recognized by Ecma International, ISO/IEC, ANSI, or other third-party standards organization, the Oracle implementation is the de facto standard.

The Oracle implementation is packaged into two different distributions: The Java Runtime Environment (JRE) which contains the parts of the Java SE platform required to run Java programs and is intended for end-users, and the Java Development Kit (JDK), which is intended for software developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.

OpenJDK is another notable Java SE implementation that is licensed under the GPL. The implementation started when Sun began releasing the Java source code under the GPL. As of Java SE 7, OpenJDK is the official Java reference implementation.

The goal of Java is to make all implementations of Java compatible. Historically, Sun's trademark license for usage of the Java brand insists that all implementations be "compatible". This resulted in a legal dispute with Microsoft after Sun claimed that the Microsoft implementation did not support RMI or JNI and had added platform-specific features of their own. Sun sued in 1997, and in 2001 won a settlement of US\$20 million, as well as a court order enforcing the terms of the license from Sun. As a result, Microsoft no longer ships Windows with Java.

Platform-independent Java is essential to Java EE, and an even more rigorous validation is required to certify an implementation. This environment enables portable server-side applications.

## **1.4 Performance**

Programs written in Java have a reputation for being slower and requiring more memory than those written in C++. However, Java programs' execution speed improved significantly with the introduction of Just-in-time compilation in 1997/1998 for Java 1.1, the addition of language features supporting better code analysis (such as inner classes, the StringBuffer class, optional assertions, etc.), and optimizations in the Java virtual machine itself, such as HotSpot becoming the default for Sun's JVM in 2000. As of December 2012, microbenchmarks show Java 7 is approximately 44% slower than C++.

Some platforms offer direct hardware support for Java; there are microcontrollers that can run Java in hardware instead of a software Java virtual machine, and ARM based processors can have hardware support for executing Java bytecode through their Jazelle option.

## **1.5 Automatic Memory Management**

Java uses an automatic garbage collector to manage memory in the object lifecycle. The programmer determines when objects are created, and the Java runtime is responsible for recovering the memory once objects are no longer in use. Once no references to an object remain, the unreachable memory becomes eligible to be freed automatically by the garbage collector. Something similar to a memory leak may still occur if a programmer's code holds a reference to an object that is no longer needed, typically when objects that are no longer needed are stored in containers that are still in use. If methods for a nonexistent object are called, a "null pointer exception" is thrown.

One of the ideas behind Java's automatic memory management model is that programmers can be spared the burden of having to perform manual memory management. In some languages, memory for the creation of objects is implicitly allocated on the stack, or explicitly allocated and deallocated from the heap. In the latter

case the responsibility of managing memory resides with the programmer. If the program does not deallocate an object, a memory leak occurs. If the program attempts to access or deallocate memory that has already been deallocated, the result is undefined and difficult to predict, and the program is likely to become unstable and/or crash. This can be partially remedied by the use of smart pointers, but these add overhead and complexity. Note that garbage collection does not prevent "logical" memory leaks, i.e. those where the memory is still referenced but never used.

Garbage collection may happen at any time. Ideally, it will occur when a program is idle. It is guaranteed to be triggered if there is insufficient free memory on the heap to allocate a new object; this can cause a program to stall momentarily. Explicit memory management is not possible in Java.

Java does not support C/C++ style pointer arithmetic, where object addresses and unsigned integers (usually long integers) can be used interchangeably. This allows the garbage collector to relocate referenced objects and ensures type safety and security.

As in C++ and some other object-oriented languages, variables of Java's primitive data types are not objects. Values of primitive types are either stored directly in fields (for objects) or on the stack (for methods) rather than on the heap, as commonly true for objects (but see Escape analysis). This was a conscious decision by Java's designers for performance reasons. Because of this, Java was not considered to be a pure object-oriented programming language. However, as of Java 5.0, autoboxing enables programmers to proceed as if primitive types were instances of their wrapper class.

Java contains multiple types of garbage collectors. By default, HotSpot uses the Concurrent Mark Sweep collector, also known as the CMS Garbage Collector. However, there are also several other garbage collectors that can be used to manage the Heap. For 90% of applications in Java, the CMS Garbage Collector is good enough.

## **1.6 Syntax**

The syntax of Java is largely derived from C++. Unlike C++, which combines the syntax for structured, generic, and object-oriented programming, Java was built almost exclusively as an object-oriented language. All code is written inside a class, and everything is an object, with the exception of the primitive data types (e.g. integers,

floating-point numbers, boolean values, and characters), which are not classes for performance reasons.

Unlike C++, Java does not support operator overloading or multiple inheritance for classes. This simplifies the language and aids in preventing potential errors and antipattern design.

Java uses similar commenting methods to C++. There are three different styles of comments: a single line style marked with two slashes (`//`), a multiple line style opened with `/*` and closed with `*/`, and the Javadoc commenting style opened with `/**` and closed with `*/`. The Javadoc style of commenting allows the user to run the Javadoc executable to compile documentation for the program.

### Example:

```
// This is an example of a single line comment using two slashes

/* This is an example of a multiple line comment using the slash and asterisk.
This type of comment can be used to hold a lot of information or deactivate
code, but it is very important to remember to close the comment. */

package fibsandlies;
import java.util.HashMap;

/**
 * This is an example of a Javadoc comment; Javadoc can compile documentation
 * from this text. Javadoc must immediately precede thing being documented.
 */
public class FibCalculator extends Fibonacci implements Calculator {
    private static HashMap<Integer, Integer> memoized = new HashMap<Integer,
Integer>();
    static {
        memoized.put(1, 1);
        memoized.put(2, 1);
    }

    /** An example of a method written in Java, wrapped in a class.
```

```

* Given a non-negative number FIBINDEX, returns
* the Nth Fibonacci number, where N equals FIBINDEX.
* @param fibIndex The index of the Fibonacci number
* @return The Fibonacci number itself
*/
public static int fibonacci(int fibIndex) {
    if (memoized.containsKey(fibIndex)) {
        return memoized.get(fibIndex);
    } else {
        int answer = fibonacci(fibIndex - 1) + fibonacci(fibIndex - 2);
        memoized.put(fibIndex, answer);
        return answer;
    }
}
}

```

## 1.7 Hello World!

The traditional Hello World program can be written in Java as:

```

class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}

```

Source files must be named after the public class they contain, appending the suffix `.java`, for example, `HelloWorldApp.java`. It must first be compiled into bytecode, using a Java compiler, producing a file named `HelloWorldApp.class`. Only then can it be executed, or 'launched'. The Java source file may only contain one public class, but it can contain multiple classes with other than public access and any number of public inner classes.

A class that is not declared public may be stored in any `.java` file. The compiler

will generate a class file for each class defined in the source file. The name of the class file is the name of the class, with `.class` appended. For class file generation, anonymous classes are treated as if their name were the concatenation of the name of their enclosing class, a `$`, and an integer.

The keyword `public` denotes that a method can be called from code in other classes, or that a class may be used by classes outside the class hierarchy. The class hierarchy is related to the name of the directory in which the `.java` file is located.

The keyword `static` in front of a method indicates a static method, which is associated only with the class and not with any specific instance of that class. Only static methods can be invoked without a reference to an object. Static methods cannot access any class members that are not also static.

The keyword `void` indicates that the main method does not return any value to the caller. If a Java program is to exit with an error code, it must call `System.exit()` explicitly.

The method name `"main"` is not a keyword in the Java language. It is simply the name of the method the Java launcher calls to pass control to the program. Java classes that run in managed environments such as applets and Enterprise JavaBean do not use or need a `main()` method. A Java program may contain multiple classes that have main methods, which means that the VM needs to be explicitly told which class to launch from.

The main method must accept an array of `String` objects. By convention, it is referenced as `args` although any other legal identifier name can be used. Since Java 5, the main method can also use variable arguments, in the form of `public static void main(String... args)`, allowing the main method to be invoked with an arbitrary number of `String` arguments. The effect of this alternate declaration is semantically identical (the `args` parameter is still an array of `String` objects), but it allows an alternative syntax for creating and passing the array.

The Java launcher launches Java by loading a given class (specified on the command line or as an attribute in a JAR) and starting its public static void `main(String[])` method. Stand-alone programs must declare this method explicitly. The `String[] args` parameter is an array of `String` objects containing any arguments passed to the class. The parameters to `main` are often passed by means of a command line.

Printing is part of a Java standard library: The `System` class defines a public



static field called out. The out object is an instance of the `PrintStream` class and provides many methods for printing data to standard out, including `println(String)` which also appends a new line to the passed string. The string "Hello, world!" is automatically converted to a `String` object by the compiler.

## 1.8 Applet

Java applets are programs that are embedded in other applications, typically in a Web page displayed in a Web browser.

```
// Hello.java  
import javax.swing.JApplet;  
import java.awt.Graphics;  
  
public class Hello extends JApplet {  
    public void paintComponent(final Graphics g) {  
        g.drawString("Hello, world!", 65, 95);  
    }  
}
```

The import statements direct the Java compiler to include the `javax.swing.JApplet` and `java.awt.Graphics` classes in the compilation. The import statement allows these classes to be referenced in the source code using the simple class name (i.e. `JApplet`) instead of the fully qualified class name (i.e. `javax.swing.JApplet`).

The `Hello` class extends (subclasses) the `JApplet` (Java Applet) class; the `JApplet` class provides the framework for the host application to display and control the lifecycle of the applet. The `JApplet` class is a `JComponent` (Java Graphical Component) which provides the applet with the capability to display a graphical user interface (GUI) and respond to user events.

The `Hello` class overrides the `paintComponent(Graphics)` method (additionally indicated with the annotation, supported as of JDK 1.5, `Override`) inherited from the `Container`

superclass to provide the code to display the applet. The `paintComponent()` method is passed a `Graphics` object that contains the graphic context used to display the applet. The `paintComponent()` method calls the graphic context `drawString(String, int, int)` method to display the "Hello, world!" string at a pixel offset of (65, 95) from the upperleft corner in the applet's display.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<!-- Hello.html -->
<html>
  <head>
    <title>Hello World Applet</title>
  </head>
  <body>
    <applet code="Hello.class" width="200" height="200">
  </applet>
  </body>
</html>
```

An applet is placed in an HTML document using the `<applet>` HTML element. The applet tag has three attributes set: `code="Hello"` specifies the name of the `JApplet` class and `width="200" height="200"` sets the pixel width and height of the applet. Applets may also be embedded in HTML using either the `object` or `embed` element, although support for these elements by Web browsers is inconsistent. However, the applet tag is deprecated, so the `object` tag is preferred where supported.

The host application, typically a Web browser, instantiates the Hello applet and creates an `AppletContext` for the applet. Once the applet has initialized itself, it is added to the AWT display hierarchy. The `paintComponent()` method is called by the AWT event dispatching thread whenever the display needs the applet to draw itself.

## 1.9 Servlet

Java Servlet technology provides Web developers with a simple, consistent mechanism for extending the functionality of a Web server and for accessing existing business systems. Servlets are server-side Java EE components that generate responses (typically HTML pages) to requests (typically HTTP requests) from clients. A servlet can almost be thought of as an applet that runs on the server side—without a face.

```
// Hello.java
import java.io.*;
import javax.servlet.*;

public class Hello extends GenericServlet {
    public void service(final ServletRequest request, final ServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        final PrintWriter pw = response.getWriter();
        try {
            pw.println("Hello, world!");
        } finally {
            pw.close();
        }
    }
}
```

The import statements direct the Java compiler to include all of the public classes and interfaces from the `java.io` and `javax.servlet` packages in the compilation.

The `Hello` class extends the `GenericServlet` class; the `GenericServlet` class provides the interface for the server to forward requests to the servlet and control the servlet's lifecycle.

The `Hello` class overrides the `service(ServletRequest, ServletResponse)` method defined by the `Servlet` interface to provide the code for the service request handler. The

service() method is passed: a ServletRequest object that contains the request from the client and a ServletResponse object used to create the response returned to the client. The service() method declares that it throws the exceptions ServletException and IOException if a problem prevents it from responding to the request.

The setContentType(String) method in the response object is called to set the MIME content type of the returned data to "text/html". The getWriter() method in the response returns a PrintWriter object that is used to write the data that is sent to the client. The println(String) method is called to write the "Hello, world!" string to the response and then the close() method is called to close the print writer, which causes the data that has been written to the stream to be returned to the client.

## 1.10 JavaServer Pages

JavaServer Pages (JSP) are server-side Java EE components that generate responses, typically HTML pages, to HTTP requests from clients. JSPs embed Java code in an HTML page by using the special delimiters <% and %>. A JSP is compiled to a Java servlet, a Java application in its own right, the first time it is accessed. After that, the generated servlet creates the response.

## 1.11 Swing Application

Swing is a graphical user interface library for the Java SE platform. It is possible to specify a different look and feel through the pluggable look and feel system of Swing. Clones of Windows, GTK+ and Motif are supplied by Sun. Apple also provides an Aqua look and feel for Mac OS X. Where prior implementations of these looks and feels may have been considered lacking, Swing in Java SE 6 addresses this problem by using more native GUI widget drawing routines of the underlying platforms.

```
// Hello.java (Java SE 5)  
import javax.swing.*;  
  
public class Hello extends JFrame {  
    public Hello() {  
        super("hello");  
    }  
}
```

```

super.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
super.add(new JLabel("Hello, world!"));
super.pack();
super.setVisible(true);
}

public static void main(final String[] args) {
    new Hello();
}
}

```

The first import includes all of the public classes and interfaces from the javax.swing package. The Hello class extends the JFrame class; the JFrame class implements a window with a title bar and a close control.

The Hello() constructor initializes the frame by first calling the superclass constructor, passing the parameter "hello", which is used as the window's title. It then calls the setDefaultCloseOperation(int) method inherited from JFrame to set the default operation when the close control on the title bar is selected to WindowConstants.EXIT\_ON\_CLOSE — this causes the JFrame to be disposed of when the frame is closed (as opposed to merely hidden), which allows the Java virtual machine to exit and the program to terminate. Next, a JLabel is created for the string "Hello, world!" and the add(Component) method inherited from the Container superclass is called to add the label to the frame. The pack() method inherited from the Window superclass is called to size the window and lay out its contents. The main() method is called by the Java virtual machine when the program starts. It instantiates a new Hello frame and causes it to be displayed by calling the setVisible(boolean) method inherited from the Component superclass with the boolean parameter true. Once the frame is displayed, exiting the main method does not cause the program to terminate because the AWT event dispatching thread remains active until all of the Swing toplevel windows have been disposed.

## **1.12 Generics**

In 2004, generics were added to the Java language, as part of J2SE 5.0. Prior to the introduction of generics, each variable declaration had to be of a specific type. For container classes, for example, this is a problem because there is no easy way to create a container that accepts only specific types of objects. Either the container operates on all subtypes of a class or interface, usually Object, or a different container class has to be created for each contained class. Generics allow compile-time type checking without having to create a large number of container classes, each containing almost identical code. In addition to enabling more efficient code, certain runtime exceptions are converted to compile-time exceptions, a characteristic known as type safety.

## **1.13 Criticism**

Criticisms directed at Java include the implementation of generics, speed, the handling of unsigned numbers, the implementation of floating-point arithmetic, and a history of security vulnerabilities in the primary Java VM implementation HotSpot.

## **1.14 Use by External Companies**

### **1.14.1 Google**

Google and Android, Inc. have chosen to use Java as a key pillar in the creation of the Android operating system, an open-source smartphone operating system. Besides the fact that the operating system, built on the Linux kernel, was written largely in C, the Android SDK uses Java to design applications for the Android platform. On May 7, 2012, a San Francisco jury found that if APIs could be copyrighted, then Google had infringed Oracle's copyrights by the use of Java in Android devices. Oracle's stance in this case has raised questions about the legal status of the language. However, the Hon. William Haskell Alsup ruled on May 31, 2012, that APIs cannot be copyrighted.

### **1.14.2 Gaikai**

Gaikai uses the Java browser plugin to stream game demos to any PC. Gaikai (Japanese for "open ocean") is a cloud-based gaming service that allows users to play high-end PC and console games via the cloud and instantly demo games and applications from a webpage on any computer or internet-connected device.

### **1.15 Documentation**

Javadoc is a comprehensive documentation system, created by Sun Microsystems, used by many Java developers. It provides developers with an organized system for documenting their code. Javadoc comments have an extra asterisk at the beginning, i.e. the tags are `/**` and `*/`, whereas the normal multi-line comments in Java are set off with the tags `/*` and `*/`.

# **LIGHTWEIGHT JAVA GAME LIBRARY**

## **2.1 Introduction**

The Lightweight Java Game Library (LWJGL) is a solution aimed directly at professional and amateur Java programmers alike to enable commercial quality games to be written in Java. LWJGL provides developers access to high performance crossplatform libraries such as OpenGL (Open Graphics Library), OpenCL (Open Computing Language) and OpenAL (Open Audio Library) allowing for state of the art 3D games and 3D sound. Additionally LWJGL provides access to controllers such as Gamepads, Steering wheel and Joysticks. All in a simple and straight forward API.

LWJGL is not meant to make writing games particularly easy; it is primarily an enabling technology which allows developers to get at resources that are simply otherwise unavailable or poorly implemented on the existing Java platform. We anticipate that the LWJGL will, through evolution and extension, become the foundation for more complete game libraries and "game engines" as they have popularly become known, and hide some of the new evils we have had to expose in the APIs.

LWJGL is available under a BSD license, which means it's open source and freely available at no charge.

## **2.2 Goals of LWJGL**

Explained by one of the major contributor.

### **2.2.1 Speed**

The whole point of LWJGL was to bring the speed of Java rendering into the 21st century. This is why we have:

Thrown out methods designed for efficient C programming that make no sense at all in java, such as glColor3fv.

Made the library throw an exception when hardware acceleration is not available on Windows. No point in running at 5fps is there?



### **2.2.2 Ubiquity**

Our library is designed to work on devices as small as phones right the way up to multiprocessor rendering servers. Just because there aren't any phones or consoles yet with fast enough JVMs and 3d acceleration is neither here nor there - there will be, one day. We're carefully tailoring the library so that when it happens we'll have OpenGL ES support in there just like that. This means that:

We had to have a very small footprint or it'll never catch on in the J2ME space at all. That's why the binary distribution is under half a meg, and that takes care of 3d sound, graphics, and IO.

Even under desktop environments having a 1-2mb download just to call a few 3D functions is daft.

We've worked to a lowest common denominator principle rather than attempting to design for all possibilities, but we've made sure that 99% of required uses are covered. That's why we've only got one window, and why we don't guarantee that windowed mode is even supported (it's officially a debug mode and hence we don't even supply some very basic windowy abilities that you'd get in AWT) and why we don't allow multiple thread rendering contexts.

### **2.2.3 Simplicity**

LWJGL needed to be simple for it to be used by a wide range of developers. We wanted relative newbies to be able to get on with it, and professionals to be able to use it professionally, maybe typically coming from a C++ background. We had to choose a paradigm that actually fits with OpenGL, and one that fits with our target platforms which ranges from PDA to desktop level. This is why:

We aren't catering for single-buffered drawing

We don't require that an instance of GL is passed around all over the place but we do not prevent this style of coding. See below for why.

We removed a lot of stuff that 99% of games programmers need to know nothing about. We have decided that consistency is better than complexity. Rather than allowing multiple ways to call the same methods and bloating the library we've just said, "Right,

no arrays. They're slower anyway. Get used to buffers, as this is what buffers are meant to be used for."

#### **2.2.4 Smallness**

See ubiquity above. We had to be small.

Small == simple. The fewer ways there are to do something, the easier it is to learn the only way that works or is allowed.

Small == our code is less buggy. Wouldn't you rather be hunting for bugs in your own code, not ours?

Small == downloadable. No version nightmares. LWJGL is small enough to download with every application that uses it.

Small == J2ME.

#### **2.2.5 Security**

We realised a few months ago that no-one was going to take us seriously if we couldn't guarantee the security of the LWJGL native libraries. This is why we:

No longer use pointers but exclusively use buffers instead

Are gradually adding further checks to buffer positions and limits to ensure that the values are within allowed ranges to prevent buffer attacks

#### **2.2.6 Robustness**

Similarly to security we have now realised that a reliable system is far more useful than a fast system. When we actually had a proper application to benchmark finally we had some real data. Many of our original design decisions were based on microbenchmarks - well, you have to start somewhere! But with a real application to benchmark we now know we can throw out asserts and replace them with a proper if (...) check and a thrown exception. We know also that we can move all that GL error checking out of native code and into Java code and we will no longer need a separate DLL for debug mode.

As for runtime exceptions, they have their place. There's not a reasonably well defined argument as to when you should use a runtime exception and when you should use a checked exception. When I made `OpenGLException` a checked exception all it did was end up littering my code with `try {} catch {}` sections - except that if you've got an `OpenGLException` there is very little sensible you can do to rectify it because it should never have occurred in the first place. That's why it's a runtime exception. You should simply not write code than can throw it because it is generally not recoverable nicely. However for robustness (and security) we are required to throw an exception if something is amiss. It falls, I believe, into exactly the same category of trouble as `NPEs`, `ArrayIndexOOBs` and `ClassCastExceptions`: should never occur but needs to be trapped somewhere.

### **2.2.7 Minimalism**

This is another critical factor in our design decisions. If it doesn't need to be in the library, it's not in the library. Our original aim was to produce a library that provided the bare minimum required to access the hardware that Java couldn't access, and by and large we're sticking to this mantra. The vector math code in the LWJGL is looking mighty scared at the moment because it's probably for the chop - well, at least, from the core library - as it's not an enabling technology at all, and there are numerous more fully featured alternatives. We chunked out `GLU` because it's mostly irrelevant to game developers except for a few functions that we really need to get redeveloped in pure Java - but basically, `GLU` is just a library of code built on top of the enablement layer.

# SLICK2D ENGINE

## 3.1 Introduction

Slick2D is an easy to use set of tools and utilities wrapped around LWJGL OpenGL bindings to make 2D Java game development easier.

Slick2D includes support for images, animations, particles, sounds, music and much much more. Additionally there are many community based projects that add additional functionality such as entity support, theme-able widgets and box2d wrappers.

Slick2D uses game containers and state based game approach as its construction.

## 3.2 Game Containers

The concept of containers isn't a particularly new one, the container provides environment in which your application (in this case a game) can run. It provides that game with a set of facilities and expects the game to comply to a certain interface. This allows the container (sometimes called the framework) to handle most of the bits of code common between all games for you - leaving you to focus on the important game logic bits and pieces.

### 3.2.1 Basic Functionality

In the case of Slick, the container holds a Game - that is the class that is contained must comply to the Game interface. To make things even easier Slick provides a basic abstract implementation of Game which you can simply extend, called BasicGame. When extending BasicGame you'll need to implement 3 methods:

`init()` - This is called when the game starts and should be used to load resources and initialise the game state.

`render()` - This method is passed a graphics content which can be used to draw to the screen. All of your game's rendering should take place in this method (or in methods called from this method)

update() - The method is called each game loop to cause your game to update it's logic. So if you have a little guy flying across the screen this is where you should make him move. This is also where you should check input and change the state of the game.

The GameContainer itself provides methods to control the properties of the game rendering and update. For instance, the game container is where you look for controlling what resolution the game runs at and whether it's in fullscreen mode. It's also responsible for maintaining the game timer and loop. There are a couple of implementations of GameContainer currently available (discussed below), however where possible you should rely on the GameContainer interface - apart from of course when constructing your container where you are making an explicit decision about how your game is to be displayed.

While the Slick game container framework is useful it doesn't suit everyone. It is intended that the features of Slick can be used outside of the framework as part of a generic LWJGL game.

### **3.2.2 Application Game Container**

The application game container is the most used, it's intended to run your game stand alone or as a webstart. It uses a simple a window to display the game and allows you to configure the display mode directly. It's generally constructed in main and started with the following lines:

```
try {
    AppGameContainer container = new AppGameContainer(new MyGame());
    container.setDisplayMode(800,600,false);
    container.start();
} catch (SlickException e) {
    e.printStackTrace();
}
```

Where the 800 and 600 specify the window resolution and the false (or true) specify whether the game should be run in fullscreen mode. Note how the container is created with an instance of the game implementation. In this way when you change containers the game itself does not need to be changed at all. This makes it very easy to make a

demo version of your game available via a webpage while maintaining the full version as a standalone application.

### 3.2.3 Applet Game Container

The applet game container uses the recently added LWJGL Applet support to present the game as a Java Applet embedded into a webpage. This functionality is currently still being tested and developed. It is known to have issues running within Opera and on low end machines. Supporting an OpenGL context within a browser window is proving to be difficult to be make stable in all cases.

However, for most people this is a great option and allows you deploy your game as an applet with the minimal of fuss. The Slick distribution provides the files required for applet distribution in the “applet” directory. To use your game in the applet you need to specify something similar to the following in the applet tag in the HTML:

```
<applet code="org.newdawn.slick.AppletGameContainer"
archive="slick.jar,testdata.jar,lwjgl_applet.jar,lwjgl.jar,lwjgl_util_applet.jar,natives.jar,j
input.jar"
width="640" height="480">
  <param name="game" value="org.newdawn.slick.tests.InputTest">
</applet>
```

Where the game parameters is replaced with the fully qualified name of your Game class and the archive list contains all the jars you require signed with an appropriate certificate. See the webstart tutorial for details on how to produce a certificate and sign the the JARs correctly.

## 3.3 State Based Games

The BasicGame class can get you a long way with simple game development. However, when games become more complicated it's often useful to separate out the different parts into separate classes with logic and rendering. These different parts are referred to states, as in the state a game is in.

In Slick this concept is supported via the Game implementation `StateBasedGame`. This game implementation proxies the render and logic methods to the “current” state. The states available and current are set externally by supplying implementations of the `GameState` interface. However, a convenience implementation of State is supplied, analogous to `BasicGame`, named `BaseGameState`.

The `GameState` is similar to the game interface, in that it has an `init`, `render` and `update` methods. However, the state based game can hold multiple of these state implementation which it can then switch between. In this way the render and logic associated with each facet of the game can be separated into different classes. As an added bonus, when swapping between these game states a visual effect can be applied to make the swap seem more fluent. These visual effects are referred to and implemented as `Transitions`.

## **3.4 Setting up Slick2D with Eclipse**

### **3.4.1 Introduction**

Since Slick2D uses LWJGL, setup is split into 2 parts, the JAR files, and the LWJGL natives (\*.dll files for Windows, \*.so files for Unix and Linux and \*.dylib/\*.jnilib for Mac).

### **3.4.2 Downloading and Extracting Slick2D and LWJGL**

Download Slick2D from [slick.ninjacave.com](http://slick.ninjacave.com)

Download LWJGL standard bundle from [lwjgl.org](http://lwjgl.org)

Extract the LWJGL zip (lwjgl-x.x.zip) file somewhere in your computer, remember or note down the location, you will need this later. We suggest you create a library (/lib) folder to store all these files in a well-known place.

### 3.4.3 Setting Up Slick2D and LWJGL in Eclipse

1. Open up Eclipse.
2. Go to *Project --> Properties* in the menu bar.
3. Click on Java Build Path.
4. Click Add Library...
5. Select User Library
6. Click Next
7. Click User Libraries
8. In the user libraries dialog select New
9. Type in Slick2D or any other name that you want for the Library Name, click ok
10. Select the new library and
  1. click the Add Jar button (if the libraries were extracted to the project space)
  2. click the Add External Jar button (if the libraries were extracted to a location outside the project space)
11. Go to where you extracted slick.zip and add the following '.jar' files ('Ctrl', or 'Shift' to select multiple entries) from 'lib' folder
  1. lwjgl.jar
  2. slick.jar
  3. jinput.jar
  4. lwjgl\_util.jar (if want to use OpenGL's GLU class)

### 3.4.4 Setting Up a Project to use LWJGL in Eclipse

In a new Java project:

1. Right-Click your project node and click Build Path > Configure Build Path or
2. Go to Project > Properties and select Build Path, select the Libraries tab
3. Click Add Library
4. Select User Library, click next
5. Add your Slick2D Library, created as instructed above
6. Setup the Native Libraries path
  1. From the Properties Dialog on the Library tab expand the Slick2D Library
    1. Click the Natives Library Location and click the Edit button



2. Navigate to the location of the LWJGL native folder and select the sub folder for the specific OS
3. Click OK
2. Alternatively before Running the project you have to add the following to the Run Configurations VM Arguments
  1. Select Run Configuration
  2. Select the Project
  3. Select the Arguments tab
  4. On VM Options put the following:

```
-Djava.library.path=<lwjgl-X.X path>/native/<linux|macosx|solaris|windows>
```

Note: Remember to select the natives of your operating system.

# MARTE ENGINE

## 4.1 Introduction

MarteEngine (ME) is a Java videogame engine with it's focus on a simple, clean API for fast game development.

Major inspiration comes from (Flashpunk) and (Slick forum).

Authors are Alberto "Gornova" Martinelli, <http://randomtower.blogspot.com> and Thomas "Tommy" Haaks, <http://www.rightanglegames.com> and Stefan "Stef569" (in order of apparence!).

Current main contributor of Marte Engine is Stef569

MarteEngine run on top of Slick Build #229 and LWJGL Version: 2.6

## 4.2 Features

### 4.2.1 Entity and World

Entity:

An Entity is everything in your game: the hero controlled by the player, some flashing information text, enemies... nearly everything. This choice was made with MarteEngine's main concept in mind: simplicity.

Entity class can (and should) be extended following the basic Java approach:

```
public class Player extends Entity { }
```

World:

World is a container for entities: imagine it as a level of a videogame. The game starts and the hero moves around in a level with obstacles, enemies, a sky and so on. The hero is just one Entity but many other entities populate your game world too: the World class permits that!

Again, creating a World is simple. Just extend MarteEngine's World class and build a basic constructor

```
public class Level extends World {
    public Level(int id, GameContainer container) {
        super(id, container);
    }
}
```

You can override the init method of World to load one or more entities into your World:

```
@Override
public void init(GameContainer container, StateBasedGame game)
    throws SlickException {
    super.init(container, game);
    Player player = new Player(100,100);
    add(player,World.GAME);
}
```

Note: you can always remove one entity using this piece of code:

```
ME.world.remove(entity);
```

This because MarteEngine store reference to current world as static variable on ME class and so you can use all world methods anywhere!

#### 4.2.2 Render and Update

Another key point to understand is that both Entity and World could update game logic and render something on screen. To do this, all of your classes that extend Entity or World can override two methods: render and update.

What does that mean in detail? Basically it means that you can render entities in any way you like (or again, let MarteEngine help you) or update game logic for particular entities (like player controlled one) in your way!

#### 4.2.3 ResourceManager

ResourceManager is an utility class: you can use it or not: it just stores references to all of our media files (spritesheets, images, sounds, font, constants) in a resource.xml file. You can write it outside of your Java classes to configure your game the way you want! In your classes, you can use your media files by just referencing them using costants!

For example let's write our Player class again using the ResourceManager to load the image:

```
public class Player extends Entity {  
    public Player(float x, float y) {  
        super(x, y);  
        // load Image using resource.xml file using constant name  
        Image img = ResourceManager.getImage("player");  
        setGraphic(img);  
    }  
}
```

Simple enough or not? You must initialize ResourceManager when your game starts using:

```
ResourceManager.loadResources("data/resources.xml");
```

And your resource.xml look like:

```
<?xml version="1.0" encoding="UTF-8"?>  
<resources>  
    <!-- basedir -->  
    <basedir path="data" />  
    <!-- sounds -->  
    <!-- songs -->  
    <!-- images -->  
    <image key="player" file="player.png" />  
    <!-- sheets -->  
    <!-- fonts -->  
</resources>
```

Syntax is (hopefully) clear! In resource.xml you define an image with key "player" where image is "player.png", so you can reference it later in your game using the "player" keyword.

#### 4.2.4 Basic Collision

For every entity you want to react on collisions, you need to declare two things:

Hitbox: The hitbox specifies the borders of an entity. When MarteEngine checks for collisions, it looks at every entity's borders if they overlap with any other hitbox,

Hitbox for wall:

```
// hitbox is a rectangle around entity, in this case it is exactly the size of the wall image
setHitBox(0, 0, img.getWidth(), img.getHeight());
// declare type of this entity
addType(SOLID);
```

Type: To speed up collision checks against different groups of entities, you can define the type of an Entity. Every entity can have many types. In our example, the Player has type "PLAYER" and the Wall has the (builtin) basic type "SOLID".

With these two basic concepts you can do anything you want.. using the collision check method for example before the player updates his/her position, it's possible to check and avoid a collision with other entities with type "SOLID":

Collision checking:

```
if (collide(SOLID, x + 10, y)==null) {
    x = x + 10;
}
```

# ECLIPSE

## 5.1 Overview

In computer programming, Eclipse is a multi-language Integrated development environment (IDE) comprising a base workspace and an extensible plug-in system for customizing the environment. It is written mostly in Java. It can be used to develop applications in Java and, by means of various plug-ins, other programming languages including Ada, C, C++, COBOL, Fortran, Haskell, JavaScript, Perl, PHP, Python, R, Ruby (including Ruby on Rails framework), Scala, Clojure, Groovy, Scheme, and Erlang. It can also be used to develop packages for the software Mathematica. Development environments include the Eclipse Java development tools (JDT) for Java and Scala, Eclipse CDT for C/C++ and Eclipse PDT for PHP, among others.

The initial codebase originated from IBM VisualAge. The Eclipse software development kit (SDK), which includes the Java development tools, is meant for Java developers. Users can extend its abilities by installing plug-ins written for the Eclipse Platform, such as development toolkits for other programming languages, and can write and contribute their own plug-in modules.

Released under the terms of the Eclipse Public License, Eclipse SDK is free and open source software (although it is incompatible with the GNU General Public License). It was one of the first IDEs to run under GNU Classpath and it runs without problems under IcedTea.

## 5.2 History

Eclipse began as an IBM Canada project. Object Technology International (OTI), which had previously marketed the Smalltalk-based VisualAge family of integrated development environment (IDE) products, developed the new product as a Java-based replacement. In November 2001, a consortium was formed with a board of stewards to further the development of Eclipse as open-source software. The original members were Borland, IBM, Merant, QNX Software Systems, Rational Software, Red Hat, SuSE,

TogetherSoft and WebGain. The number of stewards increased to over 80 by the end of 2003. In January 2004, the Eclipse Foundation was created.

Eclipse 3.0 (released on 21 June 2004) selected the OSGi Service Platform specifications as the runtime architecture.

The Association for Computing Machinery recognized Eclipse with the 2011 ACM Software Systems Award on 26 April 2012.

### **5.3 Licensing**

The Eclipse Public License (EPL) is the fundamental license under which Eclipse projects are released. Some projects require dual licensing, for which the Eclipse Distribution License (EDL) is available, although use of this license must be applied for and is considered on a case-by-case basis.

The Eclipse was originally released under the Common Public License, but was later relicensed under the Eclipse Public License. The Free Software Foundation has said that both licenses are free software licenses, but are incompatible with the GNU General Public License (GPL). Mike Milinkovich, of the Eclipse Foundation commented that moving to the GPL would be considered when version 3 of the GPL was released.

### **5.4 Name**

According to Lee Nackman, Chief Technology Officer of IBM's Rational division (originating in 2003) at that time, the name "Eclipse" (dating from at least 2001) was not a wordplay on Sun Microsystems, as the product's primary competition at the time of naming was Microsoft Visual Studio.

### **5.5 Architecture**

The Eclipse Platform uses plug-ins to provide all functionality within and on top of the runtime system, in contrast to some other applications, in which functionality is hard coded. The Eclipse Platform's runtime system is based on Equinox, an implementation of the OSGi core framework specification.

This plug-in mechanism is a lightweight software componentry framework. In addition to allowing the Eclipse Platform to be extended using other programming

languages such as C and Python, the plug-in framework allows the Eclipse Platform to work with typesetting languages like LaTeX, networking applications such as telnet and database management systems. The plug-in architecture supports writing any desired extension to the environment, such as for configuration management. Java and CVS support is provided in the Eclipse SDK, with support for other version control systems provided by third-party plug-ins.

With the exception of a small run-time kernel, everything in Eclipse is a plug-in. This means that every plug-in developed integrates with Eclipse in exactly the same way as other plug-ins; in this respect, all features are "created equal".[citation needed] Eclipse provides plug-ins for a wide variety of features, some of which are through third parties using both free and commercial models. Examples of plug-ins include a UML plug-in for Sequence and other UML diagrams, a plug-in for DB Explorer, and many others.

The Eclipse SDK includes the Eclipse Java development tools (JDT), offering an IDE with a built-in incremental Java compiler and a full model of the Java source files. This allows for advanced refactoring techniques and code analysis. The IDE also makes use of a workspace, in this case a set of metadata over a flat file space allowing external file modifications as long as the corresponding workspace "resource" is refreshed afterwards.

Eclipse implements widgets through a widget toolkit for Java called SWT, unlike most Java applications, which use the Java standard Abstract Window Toolkit (AWT) or Swing. Eclipse's user interface also uses an intermediate graphical user interface layer called JFace, which simplifies the construction of applications based on SWT.

Language packs developing by the "Babel project" provide translations into over a dozen natural languages.

## **5.6 Rich Client Platform**

Eclipse provides the Rich Client Platform (RCP) for developing general purpose applications. The following components constitute the rich client platform:



- Equinox OSGi – a standard bundling framework
- Core platform – boot Eclipse, run plug-ins[citation needed]
- Standard Widget Toolkit (SWT) – a portable widget toolkit
- JFace – viewer classes to bring model view controller programming to SWT, file buffers, text handling, text editors
- Eclipse Workbench – views, editors, perspectives, wizards

Examples of rich client applications based on Eclipse are:

- Lotus Notes 8
- Novell/NetIQ Designer for Identity Manager
- Apache Directory Studio
- Coverity Build Analysis tool.

## **5.7 Server Platform**

Eclipse supports development for Tomcat, GlassFish and many other servers and is often capable of installing the required server (for development) directly from the IDE. It supports remote debugging, allowing the user to watch variables and step through the code of an application that is running on the attached server.

## **5.8 Web Tools Platform**

The Eclipse Web Tools Platform (WTP) project is an extension of the Eclipse platform with tools for developing Web and Java EE applications. It includes source and graphical editors for a variety of languages, wizards and built-in applications to simplify development, and tools and APIs to support deploying, running, and testing apps.

## **5.9 Modeling Platform**

The Modeling project contains all the official projects of the Eclipse Foundation focusing on model-based development technologies. They are all compatible with the

Eclipse Modeling Framework created by IBM. Those projects are separated in several categories: Model Transformation, Model Development Tools, Concrete Syntax Development, Abstract Syntax Development, Technology and Research, and Amalgam

### **5.10 Model Transformation**

Model Transformation projects uses EMF based models as an input and produce either a model or text as an output. Model to model transformation projects includes ATL, an open source transformation language and toolkit used to transform a given model or to generate a new model from a given EMF model. Model to text transformation projects contains Acceleo, an implementation of MOFM2T, a standard model to text language from the OMG. Acceleo is an open source code generator that can generate any textual language (Java, PHP, Python, etc.) from EMF based models defined with any metamodel (UML, SysML, etc.).

### **5.11 Model Development Tools**

Model Development Tools projects are implementations of modeling standard used in the industry like UML or OCL and their toolkit. Among those projects can be found implementation of the following standard:

- UML
- SysML
- OCL
- BPMN
- IMM
- SBVR
- XSD

### **5.12 Extensions**

Eclipse supports a rich selection of extensions, adding support for Python via pydev, Android development via Google's ADT, JavaFX support via e(fx)clipse, and many others at the Eclipse Marketplace

## DESCRIPTION OF THE APPLICATION

### 6.1 Introduction

In this chapter I will explain what every class and how overall game works.

First, I needed to decide which tools I needed. The tools I use as follows:

GIMP Image Editor

Eclipse IDE

Light Weight Java Gaming Library

Slick2D Engine

Marte Engine

As you can see in this report there are individual chapters that gives information about for all the tools I used except GIMP image editor. Although an image editor was an important part of the creation process of this game, I though that it is not actually necessary to devote an entire chapter to describe everything about it in detail, because after all this project is not about how I made the images I use.

Having said that, before going on deep into creation process I'll like to present my reasons for choosing GIMP image editor as Image editor software.

GIMP is a freeware software that developed by community and constantly improving, getting better, it is easy to use and as good as some absurdly expensive image editor softwares.

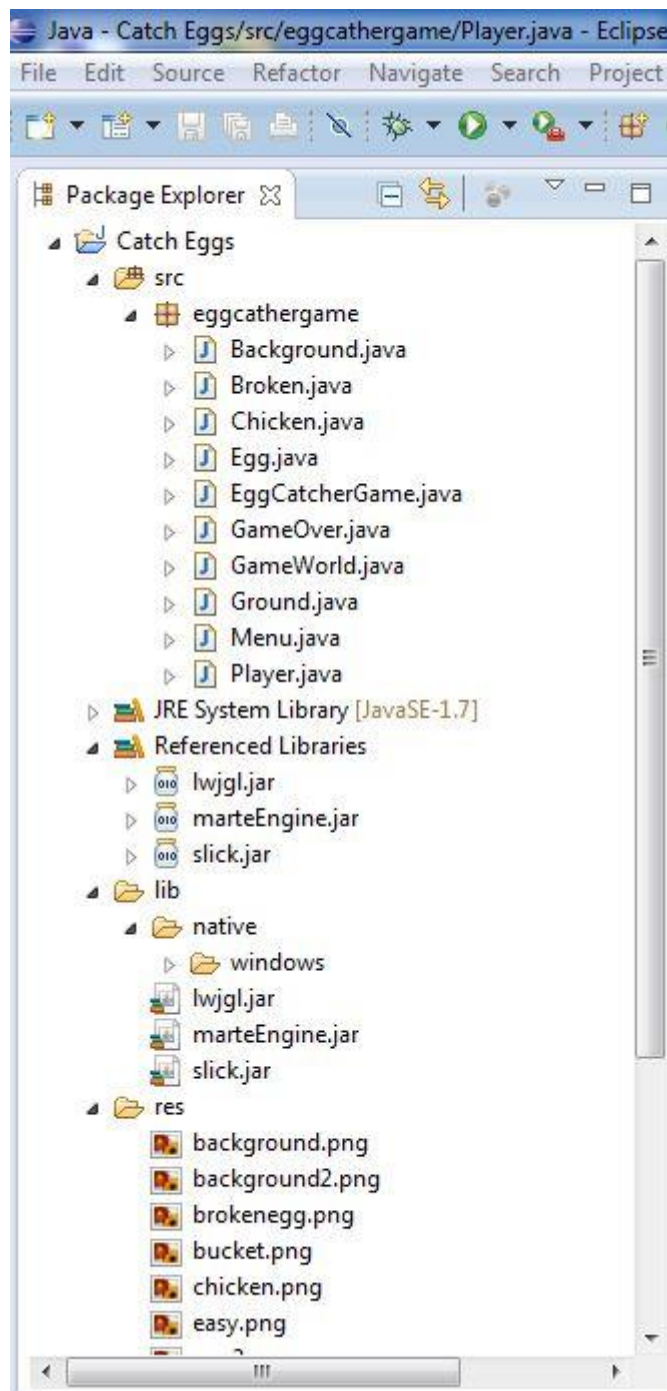
## 6.2 Concept

I've always knew that I wanted to make my own computer game as my graduation project because unlike any other area of software development while designing a computer game, developer can take on very familiar concepts and ideas that has been explored by many people before but still can made his/her own unique creation that feels and sometimes acts different then what others created using same concept and ideas.

After a long time thinking about what kind of game to make I decided to make a game in which some object would fall from sky and some other object which controlled by player would try to catch it.

After deciding on the main concept of the game I thought it would make sense if the object that suppose to fall to be an egg produced by a chicken and object that suppose to catch to be a bucket. After this point everything became clear I could see the finished game in my mind I knew how I wanted everything to be and it was time to try and make my imagination a reality on the screen.

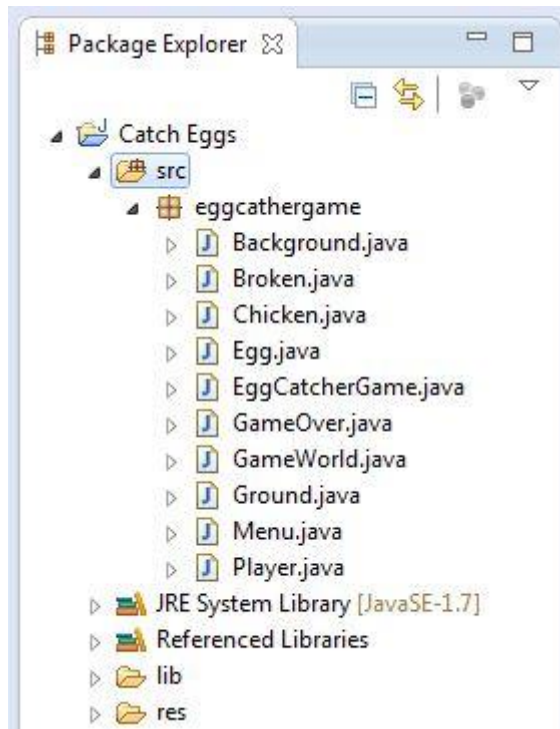
## 6.3 Project Setup



**Figure 6.1** Package Explorer

I created new Java project in eclipse and named it “Catch Eggs”. Under the “src” folder I created a source folder named “eggcathergame” to store the classes that I write. I made a folder named “lib” for storing LWJGL, Slick2D and Marte Engine Jar files and folder for images used in the game named “res”. I have setup the additional libraries following their instructions.

## 6.4 Classes

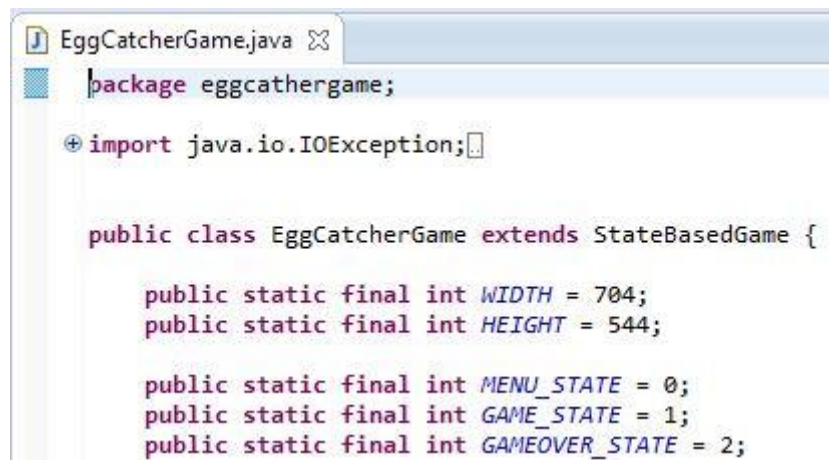


**Figure 6.2** Classes

I stored every class I write under “eggcatchergame” source folder.

I used State Based Game approach and Game Container provided with Slick2D and approached most of the elements in the game as entity provided by Marte Engine

### 6.4.1 EggCatcherGame Class



```
package eggcatchergame;

import java.io.IOException;

public class EggCatcherGame extends StateBasedGame {

    public static final int WIDTH = 704;
    public static final int HEIGHT = 544;

    public static final int MENU_STATE = 0;
    public static final int GAME_STATE = 1;
    public static final int GAMEOVER_STATE = 2;
```

**Figure 6.3** Beginning of EggCatcherGame Class

This is the main class of the project. In this class I defined that this game going to be a state based game and how many states there are going to be and how to call them, I defined Game Container and width and height of it. I have initiated resources and made it throw error message if resource initiation fails.

I made 3 states which are menu state, game state and game over state and when launched, game should start at menu state.

### 6.4.2 Menu Class

This class is the starting state. In this class I initiated images that are shown in the menu screen and made sure they shown in the exact places on the screen that I wanted.

In this class I set fps to 60 which means game screen would only refresh 60 times per second. I choose 60 because human eyes can't register more and 60 fps is more then enough for this game.

I defined four boolean for three difficulty settings named easydif, normaldif, harddif and fourth one for checking if user selected difficulty.

As the names indicate difficulties rise from easy to normal to hard. Differences between difficulties are chicken's movement speed and egg's dropping speed increase.

If users select any one of the three difficulties and press play now the state would change to game\_state and game starts but if user doesn't choose a difficulty, play now button will do nothing.

```
Menu.java
package eggcathergame;

import it.randomtower.engine.World;

public class Menu extends World {

    Image playnow, exitgame, mainpic, easyimg, normalimg, hardimg;
    public static boolean easydif = false;
    public static boolean normaldif = false;
    public static boolean harddif = false;
    public boolean difcho = false;

    public Menu(int id) {
        super(id);
    }
}
```

Figure 6.4 Beginning of Menu Class

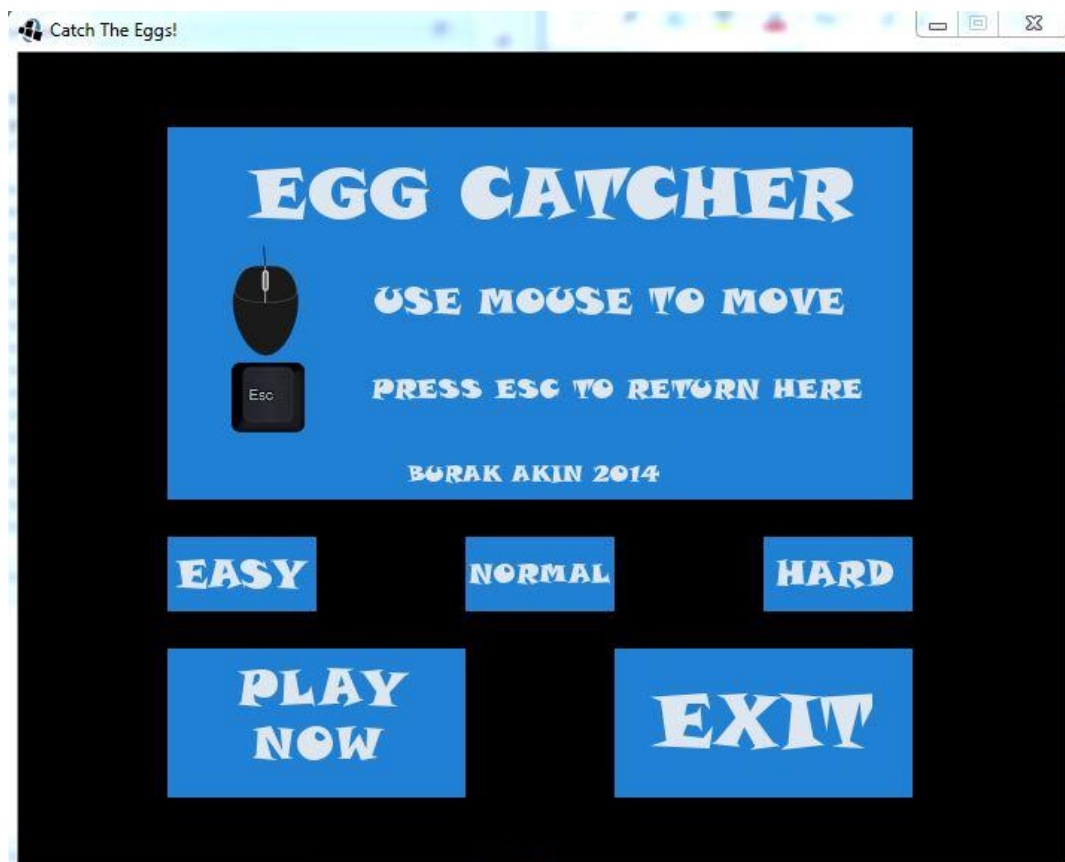


Figure 6.5 Menu Screen



### 6.4.3 GameWorld Class

This is the main game play state.

It covers where ground, chicken and bucket suppose to start on the screen.

It draws score on to the screen.

Depending if losing state triggered it changes state to game over.

If escape key is pressed go back to menu state.

```
*GameWorld.java
package eggcatchergame;

import it.randomtower.engine.World;

public class GameWorld extends World {

    public GameWorld(int id) {
        super(id);
    }

    public void init(GameContainer container, StateBasedGame sbg) throws SlickException{
```

**Figure 6.6** Beginning of GameWorld Class



**Figure 6.7** In Game Screen

#### 6.4.4 GameOver Class

This is the class that includes game over state.

This class will load when more than 3 egg breaks.

It draw Game Over message on the screen along with the score and exit button.

A screenshot of a Java IDE window titled 'GameOver.java'. The code is as follows:

```
package eggcathergame;

import it.randomtower.engine.World;

import org.lwjgl.input.Mouse;
import org.newdawn.slick.Color;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.state.StateBasedGame;

public class GameOver extends World {

    Image exitgame, over;

    public GameOver(int id) {
        super(id);
    }
}
```

**Figure 6.8** Beginning of GameOver Class



**Figure 6.9** Game Over Screen

#### 6.4.5 Background Class

Class is for background image as it's costum for Marte Engine background treated like entity.

This class draws background image on the screen.

```

Background.java
package eggcathergame;

import org.newdawn.slick.GameContainer;

public class Background extends Entity {

    public Background(float x, float y) {

        super(x, y);
        setGraphic(ResourceManager.getImage("background"));
    }
}

```

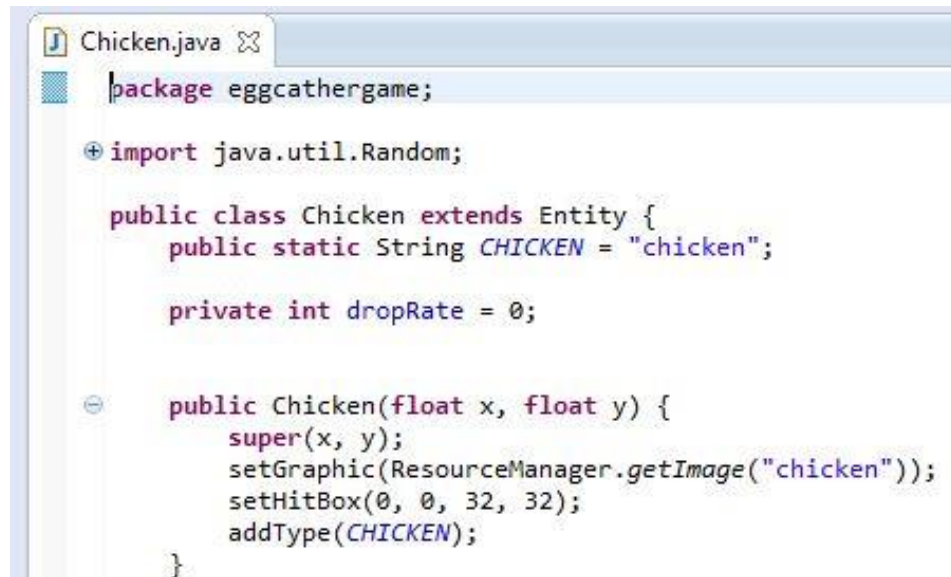
**Figure 6.10** Beginning of Background Class

### 6.4.6 Chicken Class

This class represent chicken.

It draw chicken image on the screen, set its movement to random.

Depending on the difficulty it sets drop rate of eggs and falling down speed of eggs.

A screenshot of a code editor window titled 'Chicken.java'. The code is in Java and defines a 'Chicken' class that extends an 'Entity' class. The code includes package declarations, imports, and class-level variables. The constructor initializes the chicken's position, sets its graphic, hit box, and type.

```
package eggcathergame;

import java.util.Random;

public class Chicken extends Entity {
    public static String CHICKEN = "chicken";

    private int dropRate = 0;

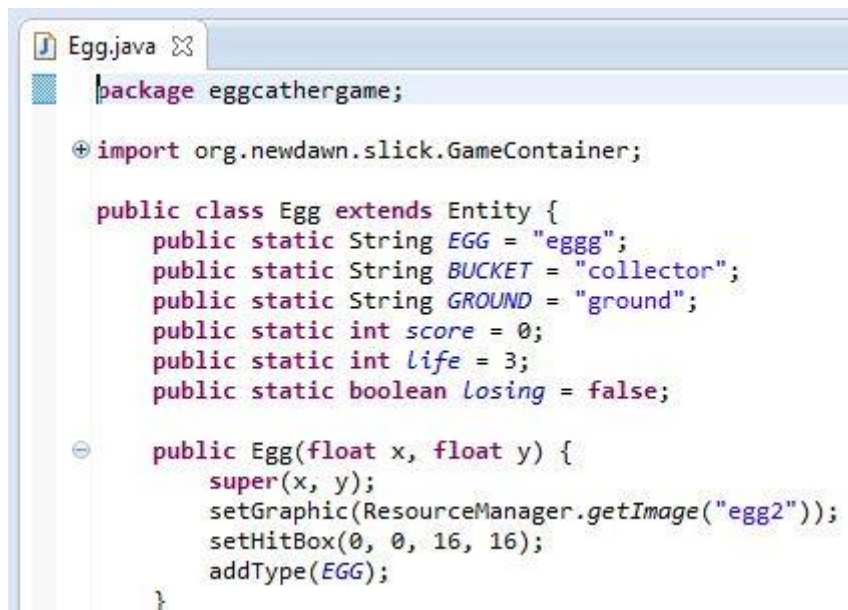
    public Chicken(float x, float y) {
        super(x, y);
        setGraphic(ResourceManager.getImage("chicken"));
        setHitBox(0, 0, 32, 32);
        addType(CHICKEN);
    }
}
```

**Figure 6.11** Beginning of Chicken Class

### 6.4.7 Egg Class

It covers the draw of egg image on the location of chicken.

Collision detection between ground and bucket. If egg collides with bucket game continuous and score increase by one and egg image remove from screen. If egg collides with ground lifes decrease by one and egg image removed from screen and broken egg image drawn instead. If egg collides with ground more then three times game is over.



```

Egg.java
package eggcathergame;

import org.newdawn.slick.GameContainer;

public class Egg extends Entity {
    public static String EGG = "eggg";
    public static String BUCKET = "collector";
    public static String GROUND = "ground";
    public static int score = 0;
    public static int life = 3;
    public static boolean losing = false;

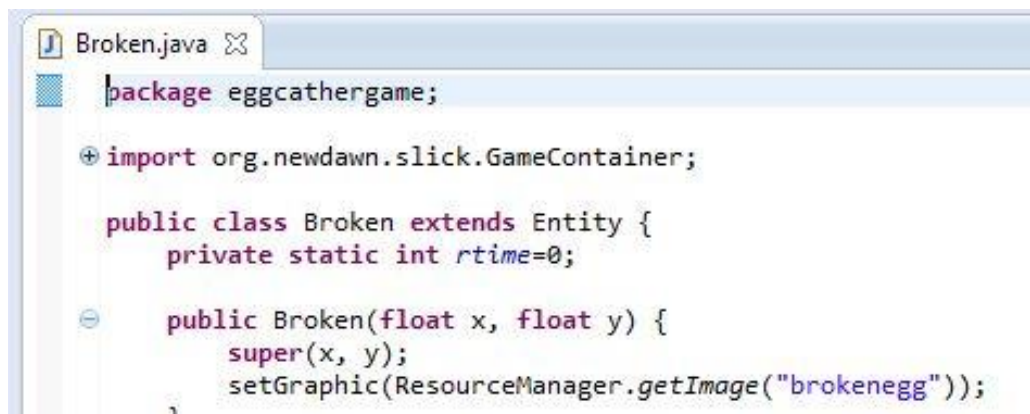
    public Egg(float x, float y) {
        super(x, y);
        setGraphic(ResourceManager.getImage("egg2"));
        setHitBox(0, 0, 16, 16);
        addType(EGG);
    }
}

```

**Figure 6.12** Beginning of Egg Class

#### 6.4.8 Broken Class

This class created to delete the broken egg images from the screen after 500milisecond.



```

Broken.java
package eggcathergame;

import org.newdawn.slick.GameContainer;

public class Broken extends Entity {
    private static int rtime=0;

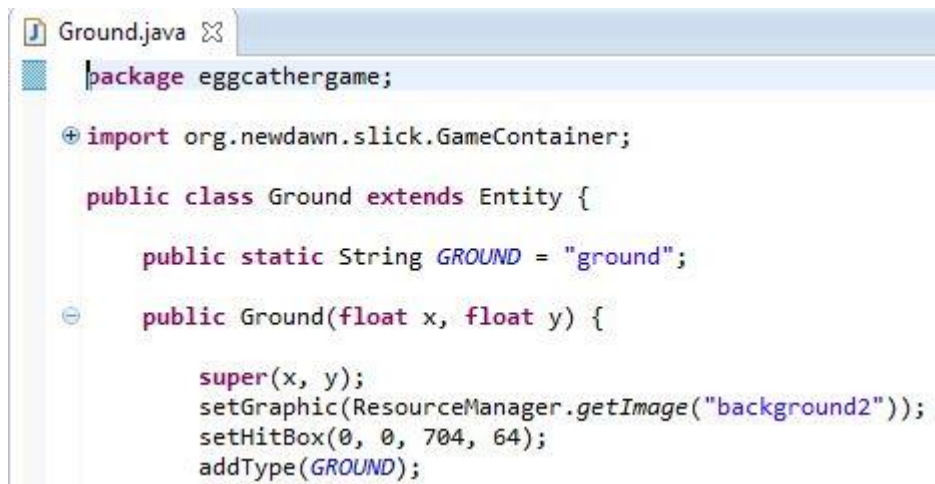
    public Broken(float x, float y) {
        super(x, y);
        setGraphic(ResourceManager.getImage("brokenegg"));
    }
}

```

**Figure 6.13** Beginning of Broken Class

#### 6.4.9 Ground Class

This class covers drawing of ground image.  
It sets hitbox and type for collision detection.



```

Ground.java
package eggcathergame;

import org.newdawn.slick.GameContainer;

public class Ground extends Entity {

    public static String GROUND = "ground";

    public Ground(float x, float y) {

        super(x, y);
        setGraphic(ResourceManager.getImage("background2"));
        setHitBox(0, 0, 704, 64);
        addType(GROUND);
    }
}


```

**Figure 6.14** Beginning of Ground Class

#### 6.4.10 Player Class

This class is for bucket.

It draws the bucket image on the screen also it covers the movement of the bucket using mouse by listening mouse movements on x direction.



```

Player.java
package eggcathergame;

import org.newdawn.slick.GameContainer;

public class Player extends Entity {

    public static String BUCKET = "collector";

    public Player(float x, float y) {

        super(x, y);
        setGraphic(ResourceManager.getImage("bucket"));
    }
}

```

**Figure 6.15** Beginning of Player Class

## CONCLUSION

Since the last decade way of distributing computer games change from physical form to online form. This change opened the doors for simple games that are developed by small developer teams or individual developers to be available to people and observation of people's purchasing choices showed that, majority of the people in the world are not necessarily interested in big, time consuming games which are developed by huge companies, but are interested in small, easy to play games.

This realization led to development of more, small but enjoyable games and with the increased interest programmers everywhere started to developing tools for speeding the coding process and for so much more.

Nowadays game industry is accepting new comers with wide as possible open arms and one can be sure of the fact that there are people who would like to spend their free time, on their own schedule playing a game which is product of ones imagination and interpretation.

The most basic reason behind the development of this game is that I wanted to turn what I imagine in my mind to reality on the screen for people to enjoy and even if I can't say everyone would enjoy it, I can say that some people will enjoy it and that makes all the effort I put in development worth it.

## REFERENCES

- [1] Java Tutorials by Oracle from the World Wide Web  
“<http://docs.oracle.com/javase/tutorial/> “
- [2] Marte Engine Documentation from the World Wide Web  
“<https://github.com/Gornova/MarteEngine/wiki>”
- [3] Marte Engine Tutorials from the World Wide Web  
“<https://github.com/Gornova/MarteEngine/wiki/Setting-up-your-enviroment>”
- [4] Slick2D Java Documents from the World Wide Web  
“<http://slick.ninjacave.com/javadoc/>”
- [5] Slick2D Game Development from the World Wide Web  
“<http://slick.ninjacave.com/wiki/>”  
“<http://slick.ninjacave.com/forum/>”
- [6] LWJGL Documentation and Tutorials from the World Wide Web  
“[http://lwjgl.org/wiki/index.php?title=Main\\_Page#Getting\\_started](http://lwjgl.org/wiki/index.php?title=Main_Page#Getting_started)”
- [7] LWJGL Java Documents from the World Wide Web  
“<http://www.lwjgl.org/javadoc/>”
- [8] Eclipse Articles, Tutorials, Demos, Books from the World Wide Web  
“<http://www.eclipse.org/resources/?category=Tutorial>”



## APPENDICES

### SOURCE CODES

#### **Background.java**

```
package eggcathergame;

import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.SlickException;

import it.randomtower.engine.ResourceManager;
import it.randomtower.engine.entity.Entity;

public class Background extends Entity {

    public Background(float x, float y) {

        super(x, y);
        setGraphic(ResourceManager.getImage("background"));

    }

    public void update(GameContainer container, int delta)
        throws SlickException {
        super.update(container, delta);
    }

    public void render(GameContainer container, Graphics g)
        throws SlickException {
        super.render(container, g);
    }

}
```

### **Broken.java**

```
package eggcathergame;

import org.newdawn.slick.GameContainer;
import org.newdawn.slick.SlickException;

import it.randomtower.engine.ME;
import it.randomtower.engine.ResourceManager;
import it.randomtower.engine.entity.Entity;

public class Broken extends Entity {
    private static int rtime=0;

    public Broken(float x, float y) {
        super(x, y);
        setGraphic(ResourceManager.getImage("brokenegg"));
    }

    public void update (GameContainer container, int delta) throws SlickException{
        super.update(container, delta);

        rtime +=delta;
        while (rtime > 500){
            ME.world.remove(this);
            rtime -=1000;
        }
    }
}
```

### **Chicken.java**

```
package eggcathergame;

import java.util.Random;
```

```

import org.newdawn.slick.GameContainer;
import org.newdawn.slick.SlickException;

import it.randomtower.engine.ME;
import it.randomtower.engine.ResourceManager;
import it.randomtower.engine.entity.Entity;

public class Chicken extends Entity {
    public static String CHICKEN = "chicken";

    private int dropRate = 0;

    public Chicken(float x, float y) {
        super(x, y);
        setGraphic(ResourceManager.getImage("chicken"));
        setHitBox(0, 0, 32, 32);
        addType(CHICKEN);
    }

    public void update (GameContainer container, int delta) throws SlickException{
        super.update(container, delta);

        Random rand = new Random();

        dropRate += delta;

        if(Menu.easydif){
            while(dropRate > 450){
                Egg eg = new Egg(x, 32);
                ME.world.add(eg);
                x =rand.nextInt(670) +16;
                dropRate -=900;
            }
        }
    }
}

```

```

        }
    }

    if(Menu.normaldif){
        while(dropRate > 400){
            Egg eg = new Egg(x, 32);
            ME.world.add(eg);
            x =rand.nextInt(670) +16;
            dropRate -=800;
        }
    }

    if(Menu.harddif){
        while(dropRate > 350){
            Egg eg = new Egg(x, 32);
            ME.world.add(eg);
            x =rand.nextInt(670) +16;
            dropRate -=700;
        }
    }
}

```

### **Egg.java**

```

package eggcathergame;

import org.newdawn.slick.GameContainer;
import org.newdawn.slick.SlickException;

import it.randomtower.engine.ME;
import it.randomtower.engine.ResourceManager;
import it.randomtower.engine.entity.Entity;

```

```

public class Egg extends Entity {
    public static String EGG = "eggg";
    public static String BUCKET = "collector";
    public static String GROUND = "ground";
    public static int score = 0;
    public static int life = 3;
    public static boolean losing = false;

    public Egg(float x, float y) {
        super(x, y);
        setGraphic(ResourceManager.getImage("egg2"));
        setHitBox(0, 0, 16, 16);
        addType(EGG);
    }

    public void update (GameContainer container, int delta) throws SlickException{
        super.update(container, delta);

        if(Menu.easydif){
            y +=(.4 * delta);
        }else if(Menu.normaldif){
            y +=(.5 * delta);
        }else{
            y +=(.6 * delta);
        }

        if (collide(GROUND, x, y) != null){
            ME.world.remove(this);
            life -=1;
            Broken bneg = new Broken(x, y+8);
            ME.world.add(bneg);
        }

        if(life == -1){

```

```

        losing = true;
    }

    if (collide(BUCKET, x, y) != null){
        score +=1;
        this.destroy();
    }
}
}

```

### **EggCatcherGame.java**

```

package eggcatchergame;

import java.io.IOException;

import it.randomtower.engine.ResourceManager;

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.state.StateBasedGame;

public class EggCatcherGame extends StateBasedGame {

    public static final int WIDTH = 704;
    public static final int HEIGHT = 544;

    public static final int MENU_STATE = 0;
    public static final int GAME_STATE = 1;
    public static final int GAMEOVER_STATE = 2;

    public static boolean ressourcesInitied = false;

```

```

private AppGameContainer container;

public EggCatcherGame() {
    super("Catch The Eggs!");
}

public void initStateList(GameContainer arg0) throws SlickException {
    if (container instanceof AppGameContainer) {
        this.container = (AppGameContainer) container;
    }
    addState(new Menu(MENU_STATE));
    addState(new GameWorld(GAME_STATE));
    addState(new GameOver(GAMEOVER_STATE));
}

public static void initRessources() throws SlickException {
    if (ressourcesInitied)
        return;
    try {
        ResourceManager.loadResources("res/resources.xml");
    } catch (IOException e) {
        e.printStackTrace();
        throw new SlickException("Resource loading failed!");
    }

    ressourcesInitied = true;
}

public static void main(String[] args) throws SlickException {
    try {

```

```

        AppGameContainer container = new AppGameContainer(new
EggCatcherGame());
        container.setDisplayMode(WIDTH, HEIGHT, false);
        container.start();
    } catch (SlickException e) {
        e.printStackTrace();
    }
}
}

```

### **GameOver.java**

```

package eggcatchergame;

import it.randomtower.engine.World;

import org.lwjgl.input.Mouse;
import org.newdawn.slick.Color;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.state.StateBasedGame;

public class GameOver extends World {

    Image exitgame, over;

    public GameOver(int id) {
        super(id);
    }

    public void init(GameContainer container, StateBasedGame sbg) throws
SlickException{
        EggCatcherGame.initRessources();
    }
}

```



```

        super.init(container, sbg);
        container.setTargetFrameRate(60);

        exitgame = new Image ("res/exitgame.png");
        over = new Image("res/over.png");

    }

    public void enter(GameContainer container, StateBasedGame sbg)
        throws SlickException {
        super.enter(container, sbg);
        this.clear();

    }

    public void render(GameContainer container, StateBasedGame sbg, Graphics g )
    throws SlickException {

        super.render(container, sbg, g);

        over.draw(100, 75);
        g.setColor(new Color (32, 129, 212));
        g.drawString("Your Score Is: "+Egg.score, 255, 300);
        exitgame.draw(250, 400);

    }

    public void update(GameContainer container, StateBasedGame sbg, int delta)
    throws SlickException{

        super.update(container, sbg, delta);

        int posX = Mouse.getX();
        int posY = Mouse.getY();

```

```

        if((posX>250 && posX<450 ) && (posY>43 && posY<146)){
            if(Mouse.isButtonDown(0)){
                System.exit(0);
            }
        }

        Mouse.setGrabbed(false);

    }

}

```

### **GameWorld.java**

```

package eggcathergame;

import it.randomtower.engine.World;

import org.lwjgl.input.Mouse;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Input;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.state.StateBasedGame;

public class GameWorld extends World {

    public GameWorld(int id) {
        super(id);
    }

    public void init(GameContainer container, StateBasedGame sbg) throws
    SlickException{

```

```

        EggCatcherGame.initRessources();
        super.init(container, sbg);

        add(new Background (0,0));
        add(new Ground (0,480));
        add(new Player (32,449));
        add(new Chicken (350,34));

    }

    public void enter(GameContainer container, StateBasedGame sbg)
        throws SlickException {
        super.enter(container, sbg);
    }

    public void render(GameContainer container, StateBasedGame sbg, Graphics g )
    throws SlickException {
        super.render(container, sbg, g);
        g.drawString("Score = "+Egg.score, 5, 0);
        g.drawString("Life = "+Egg.life, 622, 0);
    }

    public void update(GameContainer container, StateBasedGame sbg, int delta)
    throws SlickException{
        super.update(container, sbg, delta);

        if (container.getInput().isKeyPressed(Input.KEY_ESCAPE)) {
            sbg.enterState(0);
        }

        if(Egg.losing){
            sbg.enterState(2);
        }
    }

```

```

        Mouse.setGrabbed(true);

    }

}

```

### **Ground.java**

```

package eggcathergame;

import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.SlickException;

import it.randomtower.engine.ResourceManager;
import it.randomtower.engine.entity.Entity;

public class Ground extends Entity {

    public static String GROUND = "ground";

    public Ground(float x, float y) {

        super(x, y);
        setGraphic(ResourceManager.getImage("background2"));
        setHitBox(0, 0, 704, 64);
        addType(GROUND);

    }

    public void update(GameContainer container, int delta)
        throws SlickException {
        super.update(container, delta);
    }
}

```

```

        public void render(GameContainer container, Graphics g)
            throws SlickException {
            super.render(container, g);
        }
    }
}

```

### **Menu.java**

```

package eggcathergame;

import it.randomtower.engine.World;

import org.lwjgl.input.Mouse;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.state.StateBasedGame;

public class Menu extends World {

    Image playnow, exitgame, mainpic, easyimg, normalimg, hardimg;
    public static boolean easydif = false;
    public static boolean normaldif = false;
    public static boolean harddif = false;
    public boolean difcho = false;

    public Menu(int id) {
        super(id);
    }

    public void init(GameContainer container, StateBasedGame sbg) throws
    SlickException{

```

```

        //EggCatcherGame.initRessources();
        super.init(container, sbg);
        container.setTargetFrameRate(60);

        playnow = new Image("res/playnow.png");
        exitgame = new Image ("res/exitgame.png");
        mainpic = new Image("res/menuimg.png");
        easyimg = new Image("res/easy.png");
        normalimg = new Image("res/normal.png");
        hardimg = new Image("res/hard.png");

    }

    public void enter(GameContainer container, StateBasedGame sbg)
        throws SlickException {
        super.enter(container, sbg);
        this.clear();
    }

    public void render(GameContainer container, StateBasedGame sbg, Graphics g )
    throws SlickException {

        super.render(container, sbg, g);

        mainpic.draw(100, 50);
        easyimg.draw(100, 325);
        normalimg.draw(300, 325);
        hardimg.draw(500, 325);
        playnow.draw(100, 400);
        exitgame.draw(400, 400);

    }

```

```

    public void update(GameContainer container, StateBasedGame sbg, int delta)
    throws SlickException{

        super.update(container, sbg, delta);

        int posX = Mouse.getX();
        int posY = Mouse.getY();

        if((posX>100 && posX<200 ) && (posY>169 && posY<218)){
            if(Mouse.isButtonDown(0)){
                easydif=true;
                normaldif=false;
                harddif=false;
                difcho=true;
            }
        }

        if((posX>300 && posX<400 ) && (posY>169 && posY<218)){
            if(Mouse.isButtonDown(0)){
                easydif=false;
                normaldif=true;
                harddif=false;
                difcho=true;
            }
        }

        if((posX>500 && posX<600 ) && (posY>169 && posY<218)){
            if(Mouse.isButtonDown(0)){
                easydif=false;
                normaldif=false;
                harddif=true;
                difcho=true;
            }
        }
    }

```

```

        if((posX>100 && posX<300 ) && (posY>44 && posY<143)){
            if(Mouse.isButtonDown(0) && difcho){
                sbg.enterState(EggCatcherGame.GAME_STATE);
            }
        }

        if((posX>400 && posX<600 ) && (posY>44 && posY<143)){
            if(Mouse.isButtonDown(0)){
                System.exit(0);
            }
        }

        Mouse.setGrabbed(false);

    }

}

```

### **Player.java**

```

package eggcatchergame;

import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Input;
import org.newdawn.slick.SlickException;

import it.randomtower.engine.ResourceManager;
import it.randomtower.engine.entity.Entity;

public class Player extends Entity {
    public static String BUCKET = "collector";

    public Player(float x, float y) {
        super(x, y);
    }
}

```



```
        setGraphic(ResourceManager.getImage("bucket"));
        setHitBox(0, 0, 32, 32);
        addType(BUCKET);
    }

    public void update(GameContainer container, int delta) throws SlickException {
        super.update(container, delta);

        Input input = container.getInput();
        x = input.getMouseX();
    }
}
```