RESEARCH AND DEVELOPMENT (SYSTEMS)
E-GENTING SDN. BHD.

JAVA PROGRAMMING STANDARDS

First Draft
Vivian Lim
12 August 2002

TABLE OF CONTENTS

1 INTRODUCTION

This document specifies the programming standards adopted by the Research and Development (Systems) Department of E-Genting Sdn. Bhd. for the development of computer programs using the “Java” programming language.

The programming standards have been defined from the standards used in the development of the Internet games, namely the Wu Shi and FireCracker games, which both made their debut in the Internet Tournament System (ITS).

These standards exist principally to preserve the consistency of programming style in the “Java” language. Besides, they are also needed to facilitate the ongoing maintenance of the system as well as in the development of the impending Intranet Gaming System (IGS) and other future systems that will be employing the use of the “Java” programming language.

2 JAVA PROGRAMMING

2.1 General Program Layout

Each “Java” source file shall be divided into the following sections:

1. Title block,
2. Import declarations,
3. Class layout section.

Each program section and each “Java” class shall be separated from the next by a single comment line of dashes preceded and followed by blank lines.

The following sub-sections describe the layout of each section.

2.2 Program Title Block

Each “Java” source file shall be headed by a title block similar to the example showed in figure 2.1.

```
// FireSca.java - FIRECRACKER, SCALER
//
// MODULE INDEX
// NAME                 CONTENTS
// scaLdPix             Load the pixel arrays
// scaLdScs             Load scaled slides
// scaStop              Stop loading
// FireSca              Constructor
//
// MAINTENANCE HISTORY
// DATE         PROGRAMMER AND DETAILS
// 12-11-00     JS      Original
// 15-01-01     JS      Allow transparent colour in reel background
// 24-02-01     JS      Implemented Scaler super class
//
//-----------------------------------------------------------------------------
```

PROGRAM TITLE BLOCK
Figure 2.1

The title block shall contain the following sections:

1. Title line,
2. Maintenance history,
3. Module index.

Every line of the title block shall begin with a comment initiator (“//”) followed by one space. This is also applied to all the blank lines used in the title block.

The title line is the first line of the source file. It consists of the file name, a dash (“-”), followed by a short one line description of the contents of the source file in capital letters.

The “MODULE INDEX” section is only included where there is one or more methods or constructors declared under a “Java” class.

The “MODULE INDEX” section consists of a list of the names of the methods declared in the source file and one line descriptions of the methods. Method names begin at column 4 and method descriptions at column 25.

The “MAINTENANCE HISTORY” section is included in all source files. It contains a summary of changes made to the source file. The change summaries contain the following information:

COLUMN	DETAILS
4	Date of the change
17	Initials of programmer who made the change
25	Summary of the changes made

Immediately below the line “MAINTENANCE HISTORY”, the comments “DATE” starts at column 4 and “PROGRAMMER AND DETAILS” at column 17.

The first entry in “MAINTENANCE HISTORY” contains the date the program was written, the initials of the original author and the word “Original” as the description.

The title block is completed by a comment line full of dashes that begins with the comment initiator (“//”) and immediately followed by dashes (“---”) up to column 79.

2.3 Import Files Declarations

Import declarations follow the title block. They are specified in a section on their own, preceded and followed by a line full of dashes.

Every section of import files declarations is headed by a single line title in capitals letters describing the section. The single line title should be consistent in all source files, for example, if the word “IMPORTATIONS” is used, it should be used in all files.

An example of an import files declaration section is illustrated in figure 2.2.

```
//-----------------------------------------------------------------------------

// IMPORTATIONS

import java.util.*;
import java.awt.*;
import java.applet.*;
import java.awt.image.*;

//-----------------------------------------------------------------------------
```

IMPORT FILES DECLARATIONS
Figure 2.2

2.4 “Java” Class Layout

The “Java” class section comes after the import files declarations section. The “Java” class section shall contain the following sub-sections:

1. Class declaration,
2. Variable declarations,
3. Method layout,
4. Constructor layout.

An example of the format of a “Java” class layout is presented in figure 2.3.

```
//-----------------------------------------------------------------------------

// CLASS DECLARATION

class WusFrg
extends WusMep
implements WusDef, ImageObserver
{
    //-------------------------------------------------------------------------

    // STATES

    final static private int      FRGNOSL = 0;  // No slides
    final static private int      FRGRESC = 1;  // Rescale
    final static private int      FRGANIM = 2;  // Animating

    //-------------------------------------------------------------------------

    // INSTANCE DATA

    private Applet      frgApplet;      // Parent applet
    private ErrLog      frgErr;         // Error logging instance
    private WusMep      frgPrj;         // Projector instance
    private WusAni      frgAni;         // Animator instance
    private int         frgState;       // Current state
    private WusFra      frgPriFra;      // Primary frame
    private WusScs      frgScs;         // Scaled slides
    private WusFra      frgFraArr[];    // Frame array
    private int         frgFraInd;      // Frame index

    //-------------------------------------------------------------------------

    // IMAGE UPDATE CALLBACK METHOD

    public boolean
    imageUpdate (
       Image         img,          // Image being drawn
       int           flags,        // Information flags
       int           x,            // Rectangle information
       int           y,
       int           width,
       int           height)
    {
       throw new RuntimeException ("WusFrg::imageUpdate");
    }

    //-------------------------------------------------------------------------

    .
    .
    .
}
```

JAVA CLASS LAYOUT
Figure 2.3

Each "Java" class in a source file shall be separated by a blank line, followed by a line of dashes and then a class title in capital letters. The class title shall be "CLASS DECLARATION" in all cases except where the class name is different from the name of the source file. The layout for this example is illustrated in figure 2.4. A blank line then follows the class title.

The first line of the class declaration is the class name (which certainly must be the same as the file name) and any type of class modifiers (e.g. public, private, final etc.). The second line is the indication to its superclass (e.g. extends *[superclass name]*). The third line shall be the indication to its 'Interface' class or classes (e.g. implements *[interface class name]*). The example of this layout is shown in figure 2.3.

If there is no reference to any superclass, the contents of the third line shall certainly be on the second line, as shown in figure 2.4.

All class names shall begin with the uppercase and continue in lowercase. If a class name contains more than one conjoint word, the first letter of each word shall begin with a capital letter (e.g. LinkedList, Scaler, WusPrz, FireAni).

```
//-----------------------------------------------------------------------------

// PRIZE ENTRY

class WusPrz {
    WusPrz              przNxt;                 // Next prize in value order
    int                 przComb[];              // Combination
    int                 przValue;               // Prize value
}

//-----------------------------------------------------------------------------

// CLASS DECLARATION

class WusPzt
implements WusDef
{
    //-------------------------------------------------------------------------

    // INSTANCE DATA

    private WusPrz      pztPrzFst;      // First prize
    .
    .
    .
}
```

MULTIPLE CLASSES LAYOUT
Figure 2.4

Variables declarations section immediately follows the first opening of the curley braces.

Variable declarations, method declarations and constructor declarations are indented by 4 columns (a half of one standard tab stop).

Specification of the variables declarations, method declarations and constructor declarations shall be in sections 2.5, 2.6 and 2.7 respectively.

2.5 Declaration Of Instance Variables and Array Components

An example of variables declarations section is presented in figure 2.5.

```
//-------------------------------------------------------------------------

// STATES

final static private int    ANIINIT = 0;    // Initialising
final static private int    ANINOSL = 1;    // No slides
final static private int    ANIUNOPN = 2;   // Session unopened
.
.
.

//-------------------------------------------------------------------------

// INSTANCE DATA

private Applet      aniApplet;      // Parent applet
private ErrLog      aniErr;         // Error log instance
private FireReg     aniReg;         // Results generator instance
.
.
.

//-------------------------------------------------------------------------

// TEXT BORDER DISPLAY OFFSETS

final static int    aniBdrXOfs[] = {-1, -1, 0, 1, 1, 1, 0, -1};
final static int    aniBdrYOfs[] = {0, -1, -1, -1, 0, 1, 1, 1};

//-------------------------------------------------------------------------
```

DECLARATION OF CLASS DATA
Figure 2.5

This section may further be divided into 2 or more sub-sections, if there are instance variables and array components in a file. For example, in figure 2.5, the first sub-section shall be the declaration of static instance variables, the second sub-section shall be the declaration of instance variables and the third sub-section is the declaration for array components.

Each section of the variables declarations is preceded by a blank line, a line of dashes, another blank line and then a title in capital letters. The comment line full of dashes that separates each section starts at column 4 and ends at column 79.

Each variable shall be declared on a single line containing the type declaration of the variable, the name of the variable, and a comment describing the variable as indicated in figure 2.5.

Consecutive variable declarations shall be aligned on tab stops at the following point wherever possible:

1. Specification of the type of the variables,
2. Specification of the names of the variables,
3. Comments related to the variables.

Comments describing the variable should begin with a capital letter, continue in lowercase and form a clause, sentence or statement.

Specification of the variable naming conventions will be described in section 2.9.

2.6 Method Layout

Each method layout in a “Java” class shall be separated by a blank line, a line of dashes, another blank line, followed by a single title that describes the method in capital letters, and then, a blank line again. The format of the method layout shall be in accordance with the example in figure 2.6.

```
//-------------------------------------------------------------------------

// LOAD A COMBINATION INTO THE PRIZE TABLE

private void
pztLoad (
    int                 symb0,          // First symbol
    int                 symb1,          // Second symbol
    int                 symb2,          // Third symbol
    int                 value)          // Prize value
{
    WusPrz              nxt;            // Next prize table element
    WusPrz              prv;            // Previous prize table element
    WusPrz              p;              // New prize table element

    prv = null;
    for (
        nxt = pztPrzFst;
        nxt != null && nxt.przValue >= value;
        nxt = nxt.przNxt
    ) prv = nxt;
    .
    .
    .
}

//-------------------------------------------------------------------------
```

METHOD LAYOUT
Figure 2.6

The first line of the method declaration is the type of the method (e.g. void, boolean, int etc.) and any type of access modifiers (e.g. private, public or protected) together with other type of modifiers (e.g. static, final, synchronized etc.). The second line is the name of the method and the opening parenthesis of the argument. If the method has no arguments, the closing parenthesis is also placed on the second line. The arguments are then declared on successive lines indented one tab stop as shown in the example in figure 2.6. Comments for each argument are optional.

The preferred indentation levels of the argument names and comments are column 24 and 40 respectively.

If the method throws exceptions, the “throws” clause shall be placed on the line immediately after the last argument of the method, as demonstrated in figure 2.7.

```
// PROCESS AN ITS REQUEST

public Unpack
comProcReq (
    short           ttc,            // Transaction type code
    byte            bdy[],          // Body data
    int             ofs,            // Body offset
    int             len)            // Body length
throws Reject
{
    .
    .
    .
}
```

METHOD WITH THROWING EXCEPTIONS
Figure 2.7

Local variable declarations, if there are any, immediately follow the first opening braces of the method.

Local variable declarations and the outermost level of the method procedure are both indented to the first tab stop. The preferred indentation levels for local variable names and local variable comments are column 24 and 40 respectively.

Specification of the method naming conventions will be described in section 2.8.

2.7 Constructor Layout

Each constructor in a “Java” class shall be separated by a blank line, a line full of dashes, another blank line, followed by the title and then, a blank line again. The format for the constructor layout is presented in figure 2.8.

```
// CONSTRUCTOR

FireMep (
    String          name,
    ErrLog          err)            // Error logging instance

{
    mepName = name;
    mepErr = err;
    mepWaiting = false;
    mepMeq = new LinkedList ();
}
```

CONSTRUCTOR METHOD LAYOUT
Figure 2.8

The first line of the constructor declaration is the name of the constructor followed by the opening parenthesis of the argument. If the constructor has no arguments, the closing parenthesis is placed on the first line of the declaration. The arguments are then declared on consecutive lines indented one tab stop as shown in the example in figure 2.8. The indent levels for argument names and comments are the same as in method layout. Comments for each argument are optional.

2.8 Method Naming Conventions

Methods shall be named according to the class name. Each method name shall have a prefix taken from a part of its class name. All prefix characters are in lowercase and the word or words that follow the prefix will each have a capital letter for the beginning of each word. The prefix and the word or words shall be conjoined in that order to form a one-word name. For example, if the class name is “FireFrg”, a method of the class could be named “frgSendFrame”.

The word or words used after the prefix should be in accordance to the description of the methods. For example, if a method in the “Scaler” class has the description of “GRAB PIXELS”, the method shall be called “scaGrabPix”.

However, the above naming conventions do not apply to constructor and destructor methods, and other special “Java” class methods such as the “Applet” class. Constructor methods shall certainly have the same name as their classes and destructor method shall have their “Java” finalizer method, which is “finalize”.

These naming conventions are important to preserve readability and understandability of the method names. More importantly, they facilitate the use of “qref” to find all references to a method or variable in a large system.

2.9 Variable Naming Conventions

In general, variable names shall have between 1 to 18 characters.

Variables shall be named according to the class name. Each variable shall have a prefix taken from its class name. The word or words used after the prefix shall correspond to the meaning of the variables used. The prefix and the word or words shall be conjoined to create a Java identifier.

Instance variables shall have different naming conventions from local variables.

For instance variables, the prefix shall start from the second capital letter of the class name. All prefix characters shall be in lowercase and the word or words that follow the prefix will each have a capital letter for the beginning of each word. For example, if the class name is "WusMouse", an instance variable could be named "mousePoint".

For instance variables that have the "static" modifier, the prefix shall be taken from the first conjoint word of the class name. All the letters of every static variable shall be in uppercase. For example, if the class name is "FireDef", a class variable could be named "FIRESYMTOWER".

For local variables, they shall have flexible naming conventions without having to be named according to the class name.

2.10 Comments

Comments are inserted into the "Java" files in two manners.

In the case of declaration of class variables and instance variables, comments are placed in the same line as the variables, adjacent to the variable item. This example is depicted in figure 2.9.

```
    Image               fraImg;          // Frame image
    Graphics            fraGra;          // Graphics context
    Rectangle           fraBufRect;      // Buffer update rectangle
    Rectangle           fraPrjRect;      // Projection update rectangle
```

COMMENTS FOR VARIABLES
Figure 2.9

Comments for variables may also be grouped if some variables share the same comment, as shown in figure 2.10. In this example, comments are indented to the same level as the variables declarations and are preceded and followed by blank lines.

```
    // Title panel dimensions

    static final int    SCBTTLPANX = 0;         // X position
    static final int    SCBTTLPANY = 0;         // Y position
    static final int    SCBTTLPANW = 556;       // Width
    static final int    SCBTTLPANH = 60;        // Height

    // 'City of Entertainment' window dimensions

    static final int    SCBCOEWINX = 228;       // X position
    static final int    SCBCOEWINY = 5;         // Y position
    static final int    SCBCOEWINW = 100;       // Width
    static final int    SCBCOEWINH = 20;        // Height

    // 'SCOREBOARD' window dimensions

    static final int    SCBSCBWINX = 178;       // X position
    static final int    SCBSCBWINY = 22;        // Y position
    static final int    SCBSCBWINW = 200;       // Width
    static final int    SCBSCBWINH = 20;        // Height
```

GROUPED COMMENTS FOR VARIABLES
Figure 2.10

In the case of structure declarations or procedural sections, the comment is indented to the same level as the structure declaration or procedure and is preceded and followed by blank lines. This example is depicted in figure 2.11.

```
        // Calculate the display dimensions of the string

        fm = aniPriFra.fraGra.getFontMetrics ();
        width = fm.stringWidth (s);
        height = fm.getMaxAscent ();
```

COMMENTS FOR PROCEDURES
Figure 2.11

All comments shall start with a comment initiator ("//") and a blank, before the comment text. Comments shall begin with a capital letter and continue in lower or upper case. Upper case may be used for emphasis where appropriate.

2.11 Indenting

Because of the nature of the "Java" source file where the "Java" class is to be declared within the curley braces, indent levels are every 4 columns, smaller than the standard tab stop that is 8 columns.

This indent level shall be applied throughout all the "Java" files.

2.12 IF, WHILE and FOR Statement Layout

The general layouts of IF, WHILE, and FOR statements are the same. Examples of the appropriate layouts are presented in figures 2.12 to 2.15.

```
for (i = 0; i < FIREREELS; i++) itsResMod *= FIRESTOPS;
```

SINGLE CONDITIONAL STATEMENT
Figure 2.12

```
for (
    nxt = pztPrzFst;
    nxt != null && nxt.przValue >= value;
    nxt = nxt.przNxt
) prv = nxt;
```

SINGLE CONDITIONAL STATEMENT WITH LINE OVERFLOW
Figure 2.13

```
for (i = 0; i < FIREREELS; i++) {
    res.resReelPos[i] = r % FIRESTOPS;
    r /= FIRESTOPS;
}
```

MULTIPLE CONDITIONAL STATEMENT
Figure 2.14

```
if (
    (flags & ImageObserver.ERROR) != 0 ||
    (flags & ImageObserver.ALLBITS) == 0
) {
    img.flush ();
    throw new IOException ("Error while loading " + fileName);
}
```

MULTIPLE CONDITIONAL STATEMENT WITH LINE OVERFLOW
Figure 2.15

2.13 IF/ELSE Statement Layout

In the “Java” language, all IF/ELSE statements shall be written in multiple lines. The layouts for the various forms of the IF/ELSE construct are presented in figures 2.16 and 2.17.

```
if (frgScs.scsPlayBut.contains(mouse.mousePoint))
    frgAni.aniPlay ();
else if (frgScs.scsIncBetBut.contains(mouse.mousePoint))
    frgAni.aniIncBet ();
else if (frgScs.scsDecBetBut.contains(mouse.mousePoint))
    frgAni.aniDecBet ();
else if (frgScs.scsIncLinesBut.contains(mouse.mousePoint))
    frgAni.aniIncLines ();
else if (frgScs.scsDecLinesBut.contains(mouse.mousePoint))
    frgAni.aniDecLines ();
else if (frgScs.scsShowPztBut.contains(mouse.mousePoint))
    frgAni.aniShowPzt ();
else if (frgScs.scsCloseBut.contains(mouse.mousePoint))
    frgAni.aniClose ();
```

MULTIPLE LINE IF/ELSE
Figure 2.16

```
// If there is no credit left, automatically initiate
// game closure

if (aniCr <= 0) {
    aniState = ANICIP;
    aniRefresh ();
    aniReg.regClose ();
}

// If the game has been played too long, enter the
// played too long state

else if (aniRes.resTooLong) {
    aniState = ANITOOLONG;
    aniRefresh ();
}

// If further credit remains, return to the idle state

else {
    aniState = ANIIDLE;
}
```

MULTIPLE LINE IF/ELSE WITH COMMENTS
Figure 2.17

2.14 DO…WHILE Statement Layout

The layout for the DO statement is presented in figure 2.18.

```
// Select a random number within the results modulus range

do {
    r = itsGetRand();
} while (r >= itsResMod);
```

DO STATEMENT LAYOUT
Figure 2.18

2.15 SWITCH Statement Layout

The formats for the SWITCH statement are presented in figures 2.19 and 2.20.

```
switch (msg.msgMtc) {
case WusMsg.MTCOPS:         itsProcOps (msg);       break;
case WusMsg.MTCPLAY:        itsProcPlay (msg);      break;
case WusMsg.MTCCPS:         itsProcCps (msg);       break;
default:
    throw new RuntimeException ("WusIts::mepCall: bad mtc");
}
```

SWITCH STATEMENT LAYOUT – SINGLE LINE CONDITIONS
Figure 2.19

```
// Process the appropriate state transition

switch (aniState) {
case ANIINIT:
    aniState = ANIUNOPN;
    break;
case ANINOSL:
    aniState = ANIIDLE;
    aniRefresh ();
    break;
default:
    aniRefresh ();
    break;
}
```

SWITCH STATEMENT LAYOUT – MULTI LINE CONDITIONS
Figure 2.20

To prevent running out of page width, the “case” labels are always indented to the same level as the “switch” statement.

2.16 TRY…CATCH Statement Layout

The layout for the TRY…CATCH statement is presented in figure 2.21.

```
try {
    wait ();
}
catch (InterruptedException e) {
    throw new RuntimeException ("rejEdit: " + e.toString ());
}
```

TRY…CATCH STATEMENT LAYOUT
Figure 2.21

Both the “try” and “catch” blocks shall be indented to the same level.

2.17 Expression Layout

Expressions are generally laid out in free format using spaces to indicate the precedence of arithmetic or logical operations. An example of expression layout is presented in figure 2.22.

```
// Update the credit balance

funCr += res.resBasic + res.resBonus - bet*lines;
funTotWin += res.resBasic + res.resBonus;
res.resCr = funCr;
res.resTotWin = funTotWin;
```

EXPRESSION LAYOUT
Figure 2.22

3 PLAGIARISM

Plagiarism is the work of copying some other author’s work and trying to pass it as off as original. Plagiarizing other people’s work without recognition is a crime and plagiarists should be strictly punished.

In academic circles, the penalties for plagiarism range from immediate dismissal to legal action.

If a programmer uses the work of another author as the source for a new program, the original author must be acknowledged. The acknowledgement may be either be a short note in the title block or a comment in the body of the program as depicted in the examples below:

```
// MAINTENANCE HISTORY
// DATE         PROGRAMMER AND DETAILS
// 12-05-01     CK      Original, from Jonathan Searcy, FireFra.java, 13-11-00
```

EXAMPLE ACKNOWLEDGEMENT OF ORIGINAL AUTHOR IN TITLE BLOCK
Figure 3.1

```
// For each reel, calculate the reel position decrements
// to the desired stop. Adapted from Jonathan Searcy,
// FireAni.java, 14-11-00

for (i = 0; i < WUSREELS; i++) {
    startPos = aniReelOfs[i] - decs;
    while (startPos < 0) startPos += WUSSTOPS*WUSFRAMESPERSTOP;
    reqPos = aniRes.resReelPos[i] * WUSFRAMESPERSTOP;
    reelDecs = startPos - reqPos;
    if (i == 0) {
        while (reelDecs < 0)
            reelDecs += WUSSTOPS*WUSFRAMESPERSTOP;
    } else {
        while (reelDecs < 2*WUSFRAMESPERSTOP)
            reelDecs += WUSSTOPS*WUSFRAMESPERSTOP;
    }
    decs += reelDecs;
    aniFraRem[i] = decs;
}
```

ACKNOWLEDGEMENT OF ORIGINAL AUTHOR IN BODY OF PROGRAM
Figure 3.2

It is not obligatory to acknowledge the original author when copying your own work.