

Department of Computer Science

COMP1110-2004-03

Eiffel to Java for COMP1110 students

by Patrick Harrison, modified by Richard Walker

This document is an attempt to assist the transition to Java for those students who may have attended ANU courses COMP1100 or COMP1110 prior to 2004 when Eiffel was the programming language used. It is **not a good reference on Java** as in many cases it simplifies aspects of the Java language. Use it only as a guide to “get your head” around Java from what you have learned of Eiffel. Use one of the COMP1100 texts or the abundant references on the World Wide Web as references for Java. The best way to become familiar with Java’s syntax and semantics initially, **is to read as many simple Java programs as you can.**

In general, I briefly describe some aspect of Eiffel and then give a simplified explanation and/or example of how that aspect is handled in Java. Sometimes I give both Eiffel code and the Java equivalent. You will need to infer the syntax from the examples. Any references to *Lewis & Loftus* and *Riley* are to the COMP1100 text books. Sorry it is a bit long and rambling – it does not really come up to scratch as to what I had originally intended to write. I recommend you print it out rather than reading it on line. Hope it is of some use!

P. J. Harrison, February 2004

Richard Walker, July 2004

# 1 Summary

Below is a table summary of some directly comparable operations/ keywords/ examples for Java and Eiffel. Most are covered in later sections in more detail.

	JAVA	EIFFEL
single-line comments	// comment	-- comment
multiple-line comments	/* comment . . . */	no equivalent
assignment	=	:=
Integer division	num / div	num // div
modulus (remainder)	num % div	num \\ div
Boolean constants	true, false	True, False
Boolean Expression		
operators: equal	==	=
not equal	!=	/=
not	!	not
and	&&	and then
or		or else
empty reference	null	Void
object creation	x=new Class(...)	!!x.make(..)
block/multi end	}	end
var declarations	int i;	i: INTEGER
Constant declaration	final int MAX = 10;	Max: INTEGER is 10
array declarations	int[] stats;	stats: ARRAY[INTEGER]
array no of items	stats.length	stats.count
array lowest index	0	stats.lower
array highest index	stats.length-1	stats.upper
inheritance	extends ClassX	inherit ClassX
scope	public, private	{ANY}, {NONE}
Compilation	javac Class.java	compile -o class class.e make

Execution	java Class	class
Console Output	System.out.print(...)	io.put_TYPE(.....)
" " newline	System.out.println(...)	io.put_new_line
Selection	if (boolean) {...} else {...}	if boolean then...elseif boolean then... else...end
Loops	while(boolean) {...} do {...} while(boolean); for(init; test; increment) {...}	from...until...loop...end
Multiway struct	switch	inspect
Class Names convention	Example	EXAMPLE
Var Names convention	variableName	variable_name
Constants convention	A_CONSTANT	A_constant

## 2 Overview

Very much like Eiffel, Java is an object-oriented language for writing programs in which you use variables to manipulate basic data and objects. The programs are made up of classes with the executable code written in routines (called *methods* in Java) using sequence, selection, iteration, and other method calls; using dot notation to call methods and access attributes (*fields* in Java) in objects being manipulated. The precise syntax of Java is quite different from Eiffel, though it basically still consists of declarations, method headers, sequences of code and structures, etc. Note that unlike Eiffel, **Java is case sensitive**, so `maxInt` is different from the identifiers `maxint`, `Maxint` and `MaxInt`.

## 3 Source files and compilation

Eiffel classes are each coded in a separate `.e` source file and programs when compiled are assembled into a single file which is directly executable machine code, that is: a program file. The root class and all client classes making up the program are included in the single executable file which is rebuilt every time the program is compiled.

Java classes *may* be written with more than one class per `.java` source file, but for the purposes of COMP1110 this should be avoided: write one class per file with the file base name **exactly** the same name as the class name including the letter case. So, a class `CardPlayer` must be written in the file `CardPlayer.java`.

Each user-written Java class compiles into an “intermediate” `.class` file. The equivalent to Eiffel’s root class and its creation procedure is the Java class containing the `main()` method, the start of which is where program execution will begin. When compiling the Java class containing `main()` the Java compiler automatically compiles any other client classes as necessary (as for Eiffel). But in addition any user written-class can be compiled on its own. (This is not possible in Eiffel). For example, a supplier class `Card` can be compiled into the file `Card.class` without the need to compile a program which uses `Card`.

Java `.class` files are not directly executable by the operating system, but are an intermediate code which can only be run by the “Java Virtual Machine interpreter” (JVM). The `.class` file with `main()` is specified as the program and the JVM looks for the other `.class` files it needs to execute the program.

Example: A program class (with `main()` method) `A3Game` is in file `A3Game.java`. `A3Game` uses objects of some user-written supplier classes: `Card`, `CardPlayer` and `Deck`.

To compile all classes, compile the program class by entering the command line:

```
javac A3Game.java
```

This creates the intermediate code files `A3Game.class`, `Card.class`, `CardPlayer.class` and `Deck.class`.

To run the program: `java A3Game`

To recompile just the `Card` class: `javac Card.java`

Emacs has been set up with a Java menu to compile classes and run programs relatively painlessly, similar to the Eiffel menu functionality.

**Note:** Unlike Eiffel, where the source code to all the user-written supplier classes had to be provided to students in an assignment that required them to complete the `A3Game` class, in a similar Java-based assignment only a copy of the intermediate code `.class` files would be needed (and of course the details of their interface so that students would know how to use them).

## 4 Program execution start point

In Eiffel, program execution commenced at the start of the root class creation procedure. A Java program commences execution at the start of the `main()` method, which must be declared as follows:

```
public static void main(String[] args)
```

## 5 Line terminators

Eiffel statements only needed separation by white space with use of semicolons optional. In Java each program statement **must** be terminated with a semicolon. As with Eiffel, new lines, indentation, and the amount of whitespace is a human readability issue which is ignored by the compiler.

## 6 Compound/multiple statements

Eiffel uses `do...end`, `if...end`, `from...end`, `select...end`, etc. to enclose one or more code statements which belong together as a block or part of a structure.

Java uses *braces* `{ }` to enclose multiple statements or blocks of code. In many cases the braces may be omitted where there is only a single line of code to be used, but this is not recommended for beginners as it often leads to accidental omissions, especially with `if` statements. (Sun's Java style guide *requires* you to use braces in this case.)

## 7 Features/class members

Like Eiffel Features – attributes and routines, Java classes have *members* which are *fields* and *methods*. The fields record the state of an object of the class and the methods provide access to and manipulation of the fields. The order of declaration is not important, though by convention the fields, particularly those declared as `public`, are placed at the top.

## 8 Local variables

Unlike Eiffel, there is no `local` section in Java methods for declaration of local variables. Local variables may be declared anywhere in the body of code of a method, provided they are declared before use. However, it is *good*

*software engineering practice* to declare all local variables together at the top of each method. Local variables are **always** private to each routine, so just the type and identifier is required for a declaration.

```
Example: int i;
        String p1;
        Card c; CardPlayer p;
(Sun's Java style guide does not permit the last example, where two variables are declared on the same line.)
```

## 9 Variable initialization

Like Eiffel, default values for primitive data types are 0 and *false* for boolean, and *null* for all others. (Java uses *null* for what Eiffel calls *Void*, i.e., a reference which does not (yet) point to an object). In Java, both fields and local variables may optionally be initialized to a value in the same line as the declaration. Initialization of fields occurs for each object when instantiated; initialization of a method's local variables occurs each time the method is called.

```
Example: int i = 13;
        Card playingCard = new Card(2,3);
        String prompt1 = new String("Enter a value here: ");
        String prompt2 = "Enter a line of text: ";

// There are subtle differences in these two String initializations
```

## 10 Constants

Java uses a declaration modifier *final* to declare a constant identifier. Generally *final static* is used for class constants, though it is not mandatory. A final static field must be initialized with the declaration.

```
Example: Class constants (i.e., fields which are finalized)
        public final static int CARDS_PER_DEAL = 5;
        private final static String COMPANY_KEY = '234598765ABCDEF';
and for a local constant in some method:
        final int MAX_RETRIES = 200;
```

Notice the convention for naming constants: upper-case words separated by underscores. (This derives from the C/C++ convention for #define directives, because Java *final static* class constants can be used in the same way as compile time constants.)

## 11 Primitive types and objects

Somewhat like Eiffel's basic and reference types, Java has primitive data and objects. Variables of primitive data types contain the value of the primitive data, while variables of class types contain only a *reference* to an attached object. (Remember those object diagrams?) Unlike Eiffel, Java primitive data types are not objects at all and so cannot be used with dot notation to access properties or transformations of the basic value. They're just variables holding a single value.

Java's primitive types are: byte, short, int, long, float, double and boolean.

## 12 Declaring and instantiating objects

As in Eiffel, in Java there are two basic steps to take before you can use an object. First, you must declare a variable of the desired class; second, you must (*instantiate* an object and *attach* it to the variable. Creation of an object is done using the *new* operator with a *class constructor* method. Class constructors are **always the same name as the class** with zero or more arguments. There may be multiple constructors, all with the same (class) name, but they must all differ in the number or type of arguments. The new object is attached to the variable using assignment.

```
Example: Card p1, p2;           // declaration
         p1 = new Card(1,4); // card specified
         p2 = new Card();    // some default or random card
```

## 13 Routines/methods

Eiffel has *routines* which may be *procedures* or *functions*, the routine signature determining which it is. Similarly, Java's *methods* can return a value or not return a value, specified in the signature of the method. Methods which do not return a value (or reference) are declared as returning *void*.

As with Eiffel and good software practices, methods which return a value should not change the state of an object, and methods which do not return a value generally do change the state of an object, i.e., procedures are expected to change the attributes or *fields* in some way.

When you call a method in Java you must always include the parameter list in parentheses, even if there are none, for example: `s = readLine();`

So unlike Eiffel, when you read a Java program you always know whether an identifier used in an expression is a variable or a method call which returns a value, because the method call has a parameter list in parentheses.

Note: as Java was developed from C and C++, Java methods are often called "functions" whether they return a value or not, being called "void functions" when they do not return a value.

Some example Java method headers:

```
public static void main(String[] args) {

public double squareRoot(int numerator, int denominator) {

private void makeDeposit(int amount) {

private double accountBalance() {
```

## 14 Writable fields and Java's static variables

Just like Eiffel's attributes, a Java class's *fields* store the state of each object of a class. Unlike Eiffel, if the access modifier is `public`, a Java Instance Variable is **writable**, that is, it may be assigned to on the left hand side of an assignment statement. For example, a class for Cartesian coordinates might have public fields `double x, y;`. If `point1` were attached to an object of that class:

`point1.x = 10.5; point2.y = 3.5;` are legal statements.

Java also has an additional type of field declared as `static`. *Static fields*, or *Class variables* exist **only once for each class** (not a separate instance for each object) and can be accessed by using the class name instead of an object identifier: `Classname.fieldname`. The identifier used for simple text output, `System.out`, is such a variable. `System` is a class and `out` is a static field declared in the `System` class.

Similarly, classes may have *static methods* which can be called using a `Classname.methodName()` syntax without the need to create an object of that class. The easiest way to think of *statics* is that the class code itself contains exactly *one* copy and it can be accessed directly via the class name.

See *Lewis & Loftus*, page 282, or *Riley*, pages 558 & 556.

## 15 Scope: clients' access to members

Eiffel's default feature access is `{ANY}`, but it can be declared differently on a `feature {...}` line above the desired features to limit access.

Java's default access is *package*, which unfortunately can be a little tricky for the beginner. *public* access is generally applicable for many COMP1110 declarations, with *private* being used to "hide" a field or method from client use. For these, the *access modifiers* must be included in all declarations of the class, fields, and methods. See other examples in this document and either textbook, observing where `public` and `private` keywords are used in declarations.

## 16 Writable fields

Eiffel allows client classes only to read attribute values and **not** to change them directly using an assignment statement. This is a strategy of *good software design* and should be followed in well-designed Java classes by declaring fields with access `private`, and providing methods to read and update their values.

Example: My simple `CardPlayer` class that stores a name and score (see later) would be better if it included field and method declarations:

```
private int score;
public int playersScore();
public updateScore(int amount);
```

Why do all this? In the example above there may be other things that need to be done to a `CardPlayer` object when a score is changed; allowing a client class to change the score directly would lead to an inconsistent state.

## 17 Assignment and Boolean comparisons

**Assignment:** Where Eiffel uses `:=`, Java uses `=`

**Comparison:** Where Eiffel uses `=` and `/=`, Java uses `==` and `!=`

The double equals sign is often a trap for new `C/C++/Java` programmers, although it is less likely to cause a problem in Java than `C/C++`.

**Boolean Operators:** Where Eiffel uses `and` then, `or` else and `not`,

Java uses `&&`, `||` and `!`

Example: Java Assignment:

```
i = j + 4;
s = "Patrick";
opponentsCard = computer.lastCard();
```

Java Comparison:

```
if (i == MAX_TRIES) ...
while (person.score == computer.score) ...
if (first != computer) ... //note: comparing references here
while ((i < stats.length) && (stats[i] != srchvalue)) {
```

## 18 Selection: if statement

Java `if` statements only have two branches: the first part is executed if the boolean condition is true, the second if the boolean condition is false. There is no `elseif` branch in Java: you have to use nesting. Each branch of the `if` statement may consist of a single statement terminated by a semicolon or multiple statements surrounded by `{}`. The latter is recommended for beginner programmers even if there is only one statement. *Note:* there is **not** a semicolon after the boolean condition: if you put one there the compiler will interpret it as a null instruction for the true branch.

```

Example: if (i == MAX_TRIES)    // note: NO semicolon after ( boolean )
        finished = true;

        if (first == computer)
            setLeader(computer);
        else
            setLeader(person);

        if (winner == computer) {
            setLeader(computer);
            computer.updateScore(winnings);
        }
        else {
            setLeader(person);
            person.updateScore(winnings);
        }

```

Java also has the `switch` statement for multi way selection based on a single numeric value, similar to Eiffel's `inspect` statement. See the Java references for its syntax and semantics.

## 19 Iteration (loops)

Eiffel has only the one loop construct which repeatedly executes the block of statements in the loop body *until the specified condition is true*.

Java has **three** loop constructs, all of which repeatedly execute statements in their loop body *while a specified condition remains true*. Another way of saying this is that Java loops repeatedly execute the statements in the loop body until the specified condition is false!

The three Java loop constructs are functionally equivalent, and in most respects the same as in C and C++, but each lends itself better to different situations. See *Lewis & Loftus*, pages 181–183. Initialization statements are coded before each of the loop constructs, though the `for` loop includes provision in its syntax for a simple initialization statement. The discussions and examples below refer to simple counter-based loops, but apply equally to loops based on other types of exit conditions and variants.

**1) while statement:** Tests the boolean continuation condition *first*, before entering the loop body. Generally it looks like:

```

        initialization statements;
        while (this condition is true) {
            do this;
            and this etc.;
        }

```

Of course, the loop body must advance some counter or invoke some action that progresses the loop toward termination whilst maintaining the invariant. The loop body's multiple statements, including the braces `{ }`, may be replaced by a single statement (the same applies to `do` and `for` constructs):

```

        while (this condition is true)
            do this;

```

(Sun's Java style guide does not permit the omission of the braces in this way.) Notice that *there is no semicolon* directly after the `while(boolean condition)` clause.

Example: Java

```

int i;
int[] stats;
...
stats = new int[SIZE];
...
// set all stats[] items to 1
i = 0;
while (i < stats.length) {
    stats[i] = 1;
    ++i;
}

```

```

// Note: Java arrays are always
//       indexed from 0 to length - 1

```

Eiffel Equivalent

```

i: INTEGER
stats: ARRAY[INTEGER]
...
!!stats.make (0, SIZE-1)
...
-- set all 'stats' items to 1
from
    i := stats.lower
until
    i > stats.upper
loop
    stats.put (1, i)
    i := i + 1
end

```

**2) do statement** often called **do-while** statement: *Always* executes the loop body once *before testing* the boolean continuation condition. Used less frequently than the other loops as it is the least versatile. Looks like:

```

    initialization statements;
do
{
    do this;
    and this etc;
} while (this condition is true);

```

Note that in the do-while case there *is* a semicolon after the while (boolean expression); clause!

Example: Java

```

int i;
String p1 =
    "Enter number end with 0: ";
do {
    System.out.print(p1);
    i = readInteger();
    // ... do whatever with i
    // ... including i == 0
} while (i != 0);

```

Eiffel Equivalent

```

local
    i: INTEGER
    p1: STRING
do
    p1 := "Enter number end with 0: "
from
    i := 1
until
    i = 0
loop
    io.put_string (p1);
    io.read_integer
    i := io.last_integer
    -- ... do whatever with i
    -- ... including i = 0
end

```

**3) for statement** As for the while loop, tests the boolean continuation condition *first*, before entering the loop body. Most popular of the loop constructs, especially for C/C++ programmers. Syntax includes provision for a simple initialization statement(s) and counter update statement within the one for clause. Essentially the syntax is:

```

    for (initialize counter; conditional test; update counter) {
        // loop body statements to execute;
    }

```

Using the same example as the while loop above:

Example: Java

```

int i;
int[] stats;
...
stats = new int[SIZE];
...
// set all stats[] items to 1
for (i = 0; i<stats.length; ++i) {
    stats[i] = 1;
}

```

Eiffel Equivalent

```

i: INTEGER
stats: ARRAY[INTEGER]
...
!!stats.make (0, SIZE-1)
...
-- set all 'stats' items to 1
from
    i := stats.lower
until
    i > stats.upper
loop
    stats.put (1, i)
    i := i + 1
end

```

## 20 Arrays

Eiffel ARRAYs are a very flexible parameterized class with variable size and lower and upper indices, and many useful search and other routines to manipulate items in an array.

In contrast Java arrays are far more basic, though they are supposedly a normal class, albeit part of the language itself rather than from a class library. As for all reference-typed variables, an identifier for an array must be declared and then an object created and attached to it. **A Java array object's length (number of items) is specified in the creation statement and the array items are always indexed from 0 to length-1.** The length of the array object is then fixed and can not be changed.

If uninitialized, the items in arrays of primitive types are 0 (false for boolean) and in arrays of objects are *null*, i.e., do not yet reference an object (just as in Eiffel).

Arrays use `arrayname[int]` syntax for both reading and writing values at index location `int`. The size of an array is available via the `final int length` attribute. There are no other useful array methods or attributes that I am aware of. Note that the equivalent of Eiffel's `a.upper` in Java is `a.length - 1`.

Example: `int[] stats;` // declare variable to an array of int  
`stats = new int[size]` // create array object indexed 0 to size-1

`String[] prompts;` // declare an array of Strings  
`prompts = new String[];` // items are all null at this stage

`value = stats[i];` // assign item at index i to value  
`System.out.print(prompts[0]);` // print first prompt

`for (i=0; i<stats.length; i++)` // for loop clause to process  
`{ ... }` // all items using stats[i]

`int[] old;` // Create a new larger array  
`old = stats;` // object for 'stats' then  
`stats = new int[size*2];` // copy the old data items into  
`for(i=0; i<old.length; i++)` // the new array object.  
`stats[i] = old[i];`

The standard Java class library also contains many other *container classes* such as `ArrayList` and `Vector` with more features than basic arrays, but have other overheads and are not as convenient to use as basic arrays which are part of the language definition.

## 21 Creation procedures/constructors

In Eiffel, a procedure with any name can be used as a creation procedure; all that is needed is that they are nominated in the `creation` clause at the top of the class. Although we never wrote any classes with more than one creation procedure usually named `make` we used classes such as `STRING` and `STD.RAND` that had more than one creation procedure.

In Java, the equivalent mechanism is called a *constructor* method, which **is always the same name as the class**. There may be more than one with the same name, providing the parameter lists differ in number or types of arguments. Java constructors are coded without a return type (not `void`, just not mentioned at all). See the example at the end of this document.

Example: To declare a string variable and then create a string object and attach it to that variable we would:

<pre>Java String s; s = new String("String to copy from");</pre>	<pre>Eiffel s: STRING !!s.copy ("String to copy from")</pre>
------------------------------------------------------------------	--------------------------------------------------------------

Eiffel uses the creation procedure of the root class to define where a program commences execution. Java **does not use this concept**: programs commence execution at the start of the `main()` method in the class nominated on the command line to the JVM.

See the section on source files and compilation.

## 22 Assertions and exceptions

In Eiffel, assertions for routine pre- and post-conditions (*require*, *ensure*), loop invariants and variants and class invariants are built into the language and with default compilation, checked at run time. This is not the case in Java.

The general approach in Java for the above types of assertions is to include them as comments, i.e., as descriptors of what should be, or in some cases to insert custom written code. See *Riley*, page 243 and all the other pages in the index under “assertions”.

Java has extensive *exception* handling mechanisms to manage low-level errors and unusual conditions. In brief, a method may *throw* one or more exceptions if something goes wrong during its execution, for example, attempting to read a file that does not exist. Clients (i.e., from where the method is called) must either write code to *try* the method call and *catch* each exception if they occur, or pass it back up the line. Exception handling is a complex topic that is best left until you understand the structure of Java a little better.

## 23 Class library/Java API & packages

The SmartEiffel class library was relatively small, with only a few dozen classes, about a dozen which were used in COMP1100). Apart from the web pages, the `short` utility provided a comprehensive listing of the necessary information to use a class. All SmartEiffel library classes were automatically loaded by the compiler.

Java on the other hand has a fully-featured standard class library, or group of libraries really, sometimes called the *Java Application Programmer Interface* (Java API) with over 1600 classes including those necessary for *Graphical User Interface* (GUI) programming. For anything other than fairly trivial console-based programs, the API must be used extensively. There is a link on the COMP1110 Resources page to access documentation on the Java API.

The Java API has its classes organized in *packages* which are arranged in a tree structure (like a directory of files). To specify an API class we use dot notation (just like “/” in full file pathnames, but using “.”). For example, the class which you can use for random numbers is named `Random` and it is in the API package `java.util`. If you wish to use the `Random` class in your program you must either use the full package name, e.g.: `java.util.Random r;`, or you can *import* the class or all classes in the package into your program and then just use the class name.

*Importing* classes or packages does not actually bring in any code, it just lets the compiler locate the classes you are referring to. *Import statements* must be at the **top** of the class file, before your class definition, and look like:

```
import java.util.Random;    // this class can be used without qualification
import java.rmi.*;         // all classes in the java.rmi package
```

User-written classes may be organized in packages (and should be, but we will not get into that as it is a complex arrangement) and used as “user-written libraries”. The very first COMP1110 lab program uses exactly this mechanism to import a DCS-written class to simplify printing which it then uses in the code:

```
import au.edu.anu.cs.TextStreamWriter; // local package for formatted printing
```

See *Lewis & Loftus*, pages 91–95, or *Riley*, pages 72, 80.

Java also provides utilities to generate documentation, `javap` being the closest to SmallEiffel’s short listing. `javap` extracts the information from the compiled `.class` file and so only gives the signatures of fields and methods. It requires the full Java package name of the class.

```
Example: javap A4Game
         javap au.edu.anu.cs.TextStreamWriter
         javap java.lang.String
         javap java.util.Random
```

Try the last two example commands on the command line and you should see the (very) brief listing that is produced. The first example will work only if `A4Game.class` is in the current directory and the second if you are on the DCS network.

## 24 Example: simple program comparison Java – Eiffel

The `TestCardPlayer` class here is a very simple example of a main program class which creates a `CardPlayer` object, updates its score and then displays its details.

Using Eiffel the `testcardplayer.e` file is compiled with `make` specified as the root class creation routine. The compiler creates a single executable file using `TESTCARDPLAYER` and `CARDPLAYER` classes.

Using the Java compiler, `javac`, the `TestCardPlayer.java` file with `main()` method is compiled. The compiler creates the `TestCardPlayer.class` and (because it is required for the program to run) `CardPlayer.class` files. To run the program we specify `TestCardPlayer` as the class with the `main()` method by entering the command line:

```
java TestCardPlayer
```

The *Java Virtual Machine* will seek out and use the `.class` files for each of the classes used in the program.

If, say, the `CardPlayer` class in `CardPlayer.java` is changed, just that class can be recompiled using the command `javac CardPlayer.java`, and it will replace just the `CardPlayer.class` file. We can then run the program again using the changed `CardPlayer` class.

```
// Java Version of Eiffel TESTCARDPLAYER class.           // a simple test harness for CARDPLAYER class
//                                                         class TESTCARDPLAYER

// a simple test harness for CardPlayer class           creation
class TestCardPlayer                                     make
{
    public static void main(String args[]) {             feature
        CardPlayer cp;                                  make is
                                                         local
                                                         cp: CARDPLAYER
                                                         do
                                                         !!cp.make ("Patrick Harrison")
                                                         cp.update_score (100)
                                                         cp.display
                                                         end
    }
} // end TestCardPlayer class

                                                         end -- TESTCARDPLAYER class
```

## 25 Example: supplier CardPlayer class comparison Java – Eiffel

The `CardPlayer` class here is a simplified version of a class used in COMP1100 in 2003-S2, with each `CardPlayer` object having just a name and score. There are three basic Eiffel procedures: creation routine `make` to set up a newly created `CARDPLAYER` object, one to display the object’s attributes, and one to update the score. The Java

version has the equivalent fields with the same three methods: a *Constructor*, `CardPlayer` which must have the same name as the class, and the two other methods.

Please note that this is not a particularly good way to write a Java supplier class as the fields `name` and `score` are read/write to clients.

```
// Java Version of Eiffel CARDPLAYER class.
//                                     P.J.Harrison, Feb 2004

// a simplified card player class to
// record name and score

public class CardPlayer
{
    ///// Member Variables /////

    public String name; // name of the player

    public int score; // players score,initially 0

    ///// Constructor Method /////

    // pname is player's name, score 0
    public CardPlayer (String pname) {
        name = new String(pname);
        score = 0;
    }

    ///// Other Methods /////

    // print name and score of player
    public void display() {
        System.out.println(name + " score: " + score);
    }

    // update score by amount which may be negative
    public void updateScore (int amount) {
        score = score + amount;
    }

} // end of class CardPlayer

-- Eiffel CARDPLAYER class much simplified from that
-- used for 2003 COMP1100 Lab work P.J.Harrison

class CARDPLAYER
-- a simplified card player class to
-- record name and score

creation
    make

feature
    ----- Attributes -----

    name: STRING -- name of the player

    score: INTEGER --- player's score,initially 0

    ----- Creation Procedure -----

    make (pname: STRING) is
    -- pname is player's name, score 0
    do
        name := clone (pname)
        score := 0
    end

    ----- Other Routines -----

    display is
    -- print name and score of player
    do
        io.put_string (name)
        io.put_string (" score: ")
        io.put_integer (score)
        io.put_new_line
    end

    update_score (amount: INTEGER) is
    -- update score by amount which may be negative
    do
        score := score + amount
    end

end -- CARDPLAYER
```