Introduction
to

Trenton Computer Festival
May 1st & 2nd, 2004

Michael P. Redlich
Senior Research Technician
ExxonMobil Research & Engineering
michael.p.redlich@exxonmobil.com

Table of Contents

1 Introduction

This document is an introduction to the Java programming language. The engineers at Sun Microsystems modeled the language after C++ because it was an existing, popular, object-oriented programming language. However, due to high learning curve with C++, it was decided to omit its inherent complexities (pointers, templates, overloaded operators, etc.) in Java. This document will begin with how Java has evolved over the years and discuss some notable differences between C++ and Java. Since Java is an object-oriented programming language, it is important to understand the concepts of object-oriented programming. The remainder of this document will discuss object-oriented programming, Java classes and how they are implemented, introduce some keywords, and discuss some basic Java I/O.

An example Java application was developed to demonstrate the content described in this document and an upcoming ***Java Advanced Features*** document. The application encapsulates sports data such as team name, wins, losses, etc. The source code can be obtained from **`http://www.tcf-nj.org/`** or **`http://www.redlich.net/tcf/`**.

2 What is Java?

Java can be succinctly described with the following two quotes:

"Java is C++ without guns, knives, and clubs."
-- James Gosling

"Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded, dynamic language."
-- Sun Microsystems

3 Evolution of Java

Java's roots can be traced back to 1991 when a team of engineers at Sun Microsystems led by James Gosling and Patrick Naughton were interested in designing a language to be used for developing consumer devices such as cable TV boxes. The language requirements were (1) be small and generate tight code due to the lack of power and memory, (2) not be architecture-specific because manufacturers used different central processing units (CPUs), and (3) be object-oriented. The project was code named "Green." Gosling originally named this new language "Oak" because he apparently liked the look of an oak tree that was right outside his office window. However, the name, Oak, was already taken for an existing programming language. The intermediate bytecode generated from compiling this new language was designed to run on a hypothetical machine (more commonly known as a ***virtual machine***). This was the basis for the Java Virtual Machine (JVM). As a result, Java intermediate bytecode can be executed on any platform (UNIX, Intel, Mac, etc.) that had a JVM.

In 1992, the project team had difficulty marketing a remote control product called "*7." No one at Sun Microsystems was interested in developing it and consumer electronic companies weren't interested in this new technology.

The Green project was dissolved in 1994 after Naughton spent the past two years looking for buyers. Meanwhile, web technology was growing. Sun engineers developed a browser with Java, called HotJava, to display Java's power, which included the ability to interpret Java intermediate bytecode. Small Java applications were developed to add graphical animation to mostly static web pages. These small applications were the beginning of what is now known as "applets." On May 23, 1995, this "proof-of-technology" was presented at SunWorld '95, and inspired the Java craze that is still present today.

In January 1996, Netscape released version 2.0 of their browser that was Java-enabled. This was a major breakthrough for the language. Sun officially released Java 1.0 shortly after Netscape 2.0 was announced.

A major upgrade arrived in early 1997 with Java 1.1. It was the first version that made Java useful for something other than building applets. New features included:

- Java Beans
- Java Database Connectivity (JDBC)
- Reflection
- Remote Method Invocation (RMI)
- Abstract Windows Toolkit (AWT)

The next major upgrade was released in December 1998 with Java 1.2. Sun then announced a new name, Java 2, for its products such as Java 2 Standard Edition (J2SE) and Java 2 Enterprise Edition (J2EE). Therefore, Java 1.2 was essentially changed to Java 2 version 1.2. New features included:

- Java Foundation Classes (JFC)
- Consistent GUI "look-and-feel " among architectures

Java 2 version 1.3 was introduced in April 2000 and is still the present version as of {*date*}

4 Differences from C++

Since Java is modeled on the C++ programming language, it is worth mentioning some notable differences between the two languages.

Pointers

One of the most significant differences between C++ and Java is the lack of pointers. Pointers are the most common source of bugs that cause core dumps and illegal memory accesses. Because of this, the developers at Sun decided not to provide support for pointers. However, Java supports the keyword, **`this`**, as an implied ***reference*** to an object as opposed to an implied ***pointer*** in C++.

Destructors

There are **no** destructors in Java. Memory management is handled via an automatic garbage collector. It is invoked at a time the JVM deems it necessary. However, the garbage collector can be manually invoked from within an application by calling **`System.gc();`**. Writing code to manage memory is required in C++, so this is yet another complexity removed in Java.

Inheritance

C++ supports multiple inheritance where a new class can be derived from two or more base classes. Due to the complexities inherent with multiple inheritance, Java only supports single inheritance.

The syntax for defining a derived class is different as well. The examples below demonstrate those differences:

C++	Java
`class Baseball : public Sports`	`public class Baseball extends Sports`

Both statements declare the class, **`Baseball`**, to be publicly derived from the base class, **`Sports`**. The Java example introduces the keyword, **`extends`**, that is not found in C++.

Constant Member Functions (Methods)

C++ allows the developer to declare and define constant member functions (known as ***methods*** in Java and other object-oriented programming languages). A constant member function promises that data members within the current object will not be altered while the function is running. A constant member function has the signature:

```
T func(params) const; // C++ const member function signature
```

Where **`T`** is some return data type (built-in or user-defined), **`func`** is some function name, **`params`** is the parameter list, and **`const`** is the keyword placed at the end of the function declaration/definition to specify that **`func`** is a constant member function.

Java does not support constant member functions. The **`const`** keyword is not supported as well. Constant members can be defined the Java keyword, **`final`**, as in:

```
public final int i = 0;
```

Java methods can also be declared **`final`**, however this specifies that the function cannot be overridden.

Standard Template Library

Java does not include a standard template library simply because Java does not support templates. Everything is based on type **`Object`**.

Header Files

In C++, a header file is usually created to declare a class or to store a series of prototypes. A header file is referenced in the source file by using the **`#include`** compiler directive.

```
#include <cstdio>
```

directs the compiler to include the standard header **`cstdio`**. In Java, however, there are no header files. Everything is defined within a class, but classes can be grouped into a ***package***. The built-in classes included in the JDK are grouped into packages. They must be referenced from each source file as necessary using the keyword, **`import`**.

```
import java.util.*;
```

declares that all classes within the **`java.util`** package are available.

Access Specifiers

Java supports the same access specifiers used in C++, i.e., the keywords, **`public`**, **`protected`**, and **`private`**. However, the use of these specifiers is slightly different.

In C++, blocks of data members and member functions are declared to have the specified access. Each block begins with the access specifier keyword followed by a colon (**`:`**). If an access specifier is not declared, **`private`** is the default access specifier. The following example demonstrates the use of access specifiers in C++:

```
class Sports
    {
    private:
        int win;
        ...
    public:
        Sports(void);
        Sports(int,int,int);
        ~Sports(void);
        int getWin(void) const;
        ...
    };
```

In Java, data members and methods are individually declared with the appropriate access specifier as demonstrated in the following example:

```
public class Sports
    {
    private int win;
    ...
    public Sports(int w,int l,int t)
        {
        ...
        }
    ...
    public int getWin()
        {
        return win;
        }
    }
```

If an access specifier is not declared on an individual data member or method, it has default package access, i.e., access is extended to other classes within a particular package.

Use of **`void`** Keyword

As with C++, the keyword, **`void`**, can still be used as a return type in Java. However, unlike C++, it is an error to use **`void`** to declare empty function or class parameter lists. For example:

```
public int getWin(void) // error
    {
    return win;
    }

public int getWin() // OK
    {
    return win;
    }
```

5 Object-Oriented Programming

*Please note this chapter is the same as the corresponding Object-Oriented Programming chapter of the **Introduction to C++** document.*

Programming Paradigms

There are two programming paradigms:
- Procedure-Oriented
- Object-Oriented

Examples of procedure-oriented languages include:
- C
- Pascal
- FORTRAN

Examples of object-oriented languages include:
- Java
- C++
- SmallTalk
- Eiffel

A side-by-side comparison of the two programming paradigms clearly shows how object-oriented programming is vastly different from the more conventional means of programming:

Procedure-Oriented Programming	**Object-Oriented Programming**
• Top Down/Bottom Up Design	• Identify objects to be modeled
• Structured programming	• Concentrate on what an object does
• Centered around an algorithm	• Hide how an object performs its tasks
• Identify tasks; how something is done	• Identify an object's behavior and attributes

Some Object-Oriented Programming (OOP) Definitions

An ***abstract data type*** (ADT) is a user-defined data type where objects of that data type are used through provided functions without knowing the internal representation. For example, an ADT is analogous to, say an automobile transmission. The car's driver knows how to operate the transmission, but does not know how the transmission works internally.

The ***interface*** is a set of functions within the ADT that allow access to data.

The ***implementation*** of an ADT is the underlying data structure(s) used to store data.

It is important to understand the distinction between a ***class*** and an ***object***. The two terms are often used interchangeably, however there are noteworthy differences. Classes will be formally introduced later in this document, but is mentioned here due to the frequent use of the nomenclature in describing OOP. The differences are summarized below:

Class	**Object**
• Defines a model	• An instance of a class
• Declares attributes	• Has state
• Declares behavior	• Has behavior
• An ADT	• There can be many ***unique*** objects of the same class

Main Attributes of OOP

There are four main attributes to object-oriented programming:

- Data Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism

Data Encapsulation

Data encapsulation separates the implementation from the interface. User access to data is only allowed through a defined interface. Data encapsulation combines information and an object's behavior.

Data Abstraction

Data abstraction defines a data type by its functionality as opposed to its implementation. For example, the protocol to use a double-linked list is made public through the supplied interface. Knowledge of the implementation is unnecessary and therefore hidden.

Inheritance

Inheritance is a means for defining a new class as an extension of a previously defined class. A ***derived*** class **inherits** all attributes and behavior of a ***base*** class, i.e., it provides access to all data members and member functions of the base class, and allows additional members and member functions to be added if necessary.

The base class and derived class have an "is a" relationship. For example,

- Baseball (a derived class) ***is a*** Sport (a base class)
- Pontiac (a derived class) ***is a*** Car (a base class)

Polymorphism

Polymorphism is the ability of different objects to respond differently to virtually the same function. For example, a base class provides a function to print the current contents of an object. Through inheritance, a derived class can use the same function without explicitly defining its own. However, if the derived class must print the contents of an object differently than the base class, it can override the base class's function definition with its own definition. In order to invoke polymorphism, the function's return type and parameter list must be identical. Otherwise, the compiler ignores polymorphism.

Polymorphism is derived from the Greek meaning "many forms." It is a mechanism provided by an object-oriented programming language, rather than a programmer-provided workaround.

Advantages of OOP

- The implementation of an ADT can be refined and improved without having to change the interface, i.e., existing code within an application doesn't have to be modified to accommodate changes in the implementation.
- Encourages modularity in application development.
- Better maintainability of code yielding less code "spaghetti."
- Existing code can be reused in other applications.

6 Some Java Keywords

The keywords defined below are just a subset of the complete C++ keyword list.

- **`class`** – used for declaring/defining a class.
- **`new`** – allocate storage on the free store (heap). Note: no corresponding **`delete`** keyword
- **`private`**/**`protected`**/**`public`** – access specifiers used for data hiding which is a means of protecting data.
  - **`private`** – not visible outside of the class.
  - **`protected`** – like private except visible only to derived classes through inheritance.
  - **`public`** – visible to all applications.
- **`try`**/**`throw`**/**`catch`** – used in exception handling.
- **`final`** – prevents a class from being a parent class or prevents a method within a class from being overridden.
- **`extends`** – defines a derived class from the specified base class.
- **`abstract`** – a declaration specifier for a class or method to indicate an implementation doesn't exist.
- **`implements`** – specifies that a particular interface will be implemented in a particular class.
- **`boolean`**/**`false`**/**`true`** – used for Boolean logic.
  - **`boolean`** – data type that can only accept the values **`true`** and **`false`**.
  - **`false`** – numerically zero.
  - **`true`** – numerically one.

7 Basic Java I/O

The most basic Java I/O is writing to and reading from the console. Java contains the built-in **`System`**, **`InputStream`**, and **`PrintStream`** classes to perform these operations. To read from the keyboard, the statement

```
System.in.read();
```

will read a sequence of characters until the end of stream, i.e., the ***Enter*** key, has been detected or an exception has been thrown.

To write data to the screen, the statements:

```
System.out.print("Hello, world!");
System.out.println("Hello, world!");
```

will print the string "Hello, world!" on the screen. The difference between the two is that the latter will automatically append a carriage return/line feed to the string.

Note that this is for Java ***applications*** only. A Java ***applet*** must use a **`Graphics`** object for displaying text and other graphics on the screen. More detailed information regarding Java I/O will be available in an upcoming ***Java Advanced Features*** document.

8 Java Classes

As mentioned earlier, a Java class is a user-defined ADT. It encapsulates a data type and any operations on it. By default, data members and methods in a class have ***package*** access if the usual access specifiers (**`private`**, **`protected`**, and **`public`**) are not explicitly specified. An ***abstract class*** is one that declares at least one abstract method along with regular methods and data members.

A basic Java class usually contains the following elements:

- Constructor(s) – creates an object.
- Data members – object attributes.
- Methods – operations on the attributes.

Unlike C++, ***everything*** in Java must be wrapped within a class. This means that an application cannot have a source file containing code without classes. A Java application starts execution from a function called **`main`**, but it must be wrapped within a class. This **`main`** function must be static and ***always*** contains the Java equivalent of a C++ parameter list.

Each one of these is demonstrated in a simple example:

```
public class Sports
    {
    // private data members:
    private String team;
    private int win;
    private int loss;

    // primary constructor
    public Sports(String team,int win,int loss)
        {
        setTeam(team);
        setWin(win);
        setLoss(loss);
        }

    // methods
    public String getTeam()
        {
        return team;
        }

    public void setTeam(String team)
        {
        this.team = team;
        }

    public int getWin()
            {
            return win;
            }

    public void setWin(int win)
        {
        this.win = win;
        }

    ...
    }
```

```
public class SportsApp
    {
    public static void main(String[] args)
        {
        Sports sports = new Sports("Mets",94,68);
        System.out.println(sports.getTeam());
        }
    }
```

Note the use of a separate class for function **`main`**. The function could have easily been placed within the definition of class **`Sports`**. Placing main in a separate function distinguishes where the application begins from the classes that support the application. Also note the use of the keyword, **`this`** within the **`setTeam()`** and **`setWin()`** methods. It allows using the same data member variable names in the method parameter list. Therefore,

```
this.win = win;
```

Means assign the variable, **`win`**, in the parameter list of **`public void setWin(int win)`** to the class data member variable, **`win`**.

Java comments are the same as C++ comments, i.e., they begin with a double slash (**`//`**). Anything after a double slash until the end of the current line is considered a comment by the compiler. C comments (**`/* ... */`**) can still be used in a Java application as well.

Note that constructors have the same name as the class and have **no** return type.

More than one constructor can be written for a particular class. The different constructor types are:

- Default constructors
- Primary constructors

Default Constructors

A ***default constructor*** creates objects with specified default values. A default constructor added to **`Sports`** might look like:

```
// constructor
public Sports()
    {
    setTeam("Team");
    setWin(0);
    setLoss(0);
    }
```

Unlike C++, there is **no** automatic default constructor generated if one is not explicitly defined.

Primary Constructors

A ***primary constructor*** creates objects with the argument values passed in the constructor parameter list. More than one primary constructor may be defined for a class. The primary constructor in **`Sports`** is defined as:

```
// primary constructor
Sports(String team,int win,int loss)
    {
    ...
    }
```

If the application requires, say, a single String object in the parameter list, then a second constructor can be defined as:

```
// another primary constructor
Sports(String team)
    {
    ...
    }
```

9 Class Instantiation

C++ classes can be instantiated both statically and dynamically. In Java, however, there is only one way to instantiate classes due to the lack of pointers. For example, consider a **`Baseball`** class that is derived from **`Sports`**. It has the following constructor:

```
public class Baseball(String team,int win,int loss)
    {
    ...
    }
```

Dynamic instantiation in C++ required using the keyword **`new`**. This same keyword is used to instantiate Java classes as well. Below is a comparison of class instantiation between C++ and Java:

C++

```
Baseball *bball = new Baseball("Mets",94,68) // dynamic memory allocation
Baseball bball("Mets",94,68); // static memory allocation
```

Java

```
Baseball bball = new Baseball("Mets",94,68);
```

All three statements above declare **`bball`** as an object of type **`Baseball`** except for the first C++ statement, which declares **`bball`** as a pointer to an object of type **`Baseball`**. All objects contain the values **`"Mets"`**, **`94`**, and **`68`**.

Once the object is created, any public member functions are called using the name of the object and the structure dot operator (**`.`**). For example,

```
bball.getWin();
```

calls the function **`getWin()`**. The object remains alive until the garbage collector determines the object is not in use. This can be accomplished manually by assigning the name of the object to **`null`**. This removes the reference attached to the object.

```
bball = null;
```

It can be seen that Java uses a combination of C++'s two methods of instantiating classes.

10 Popular Java Integrated Development Environments

Sun Microsystems offers the Java Development Kit (JDK) from their Java web site (http://java.sun.com/) for no charge. It is a command line environment, but it is guaranteed to be portable. Comprehensive HTML documentation can be obtained separately from the web site as well. However, some vendors have licensed the JDK to sell an integrated development environment (IDE) package. These vendors have added their own additional libraries for faster applications development. However, portability is now limited. The most common vendors offering Java IDEs are:

- Borland JBuilder
  - **`http://www.borland.com/`**
- Symantec Visual Cafe
  - **`http://www.symantec.com/`**
- IBM Visual Age
  - **`http://www.ibm.com/`**
- Microsoft Visual J++
  - **`http://www.microsoft.com/`**

11 References for Further Reading

The references listed below are only a small sampling of resources where further information on Java can be obtained:

- Thinking in Java (*book*)
  - Bruce Eckel
  - ISBN
  - **`http://www.bruceeckel.com/`**
- Java Developer's Journal (*monthly periodical*)
  - **`http://www.javadevelopersjournal.com/`**
- Core Java 2, Volume I - Fundamentals (*book*)
  - Cay S. Horstmann and Gary Cornell
  - ISBN 0-13-081933-6
  - **`http://www.sun.com/books/catalog/horstmann6/`**
- Core Java 2, Volume II - Advanced Features (*book*)
  - Cay S. Horstmann and Gary Cornell
  - ISBN 0-13-081934-4
  - **`http://www.sun.com/books/catalog/horstmann7/`**
- The Java Tutorial for the Real World (*book*)
  - Yakov Fain
  - ISBN 0-9718439-0-2
  - **`http://www.smartdataprocessing.com/`**

12 References

- **`java.sun.com`** web site.
- Cay S. Horstmann and Gary Cornell. *Core Java 1.1*, ISBN 0-13-766957-7.
- Bruce Eckel. *Thinking in Java*, ISBN 0-13-027363-5.