SAGE Computing Services

All you (ever) needed to know about Java

0419 904 458
www.sagecomputing.com.au

All you (ever) needed to know about Java Workshop

Edition 1.0 November 2006

SAGE Computing Services

A Company Profile

Telephone: 0419 904 458
Facsimile: 08 9243 4335
www.sagecomputing.com.au

Services

Services provided by the company include:

- Customised Training Programs
- Post-training Mentoring
- Database Administration Services
- Complete Systems Development and Support
- Consultancy Advice
- Quality Assurance
- Tuning Reviews
- Business Analysis
- Telephone Advice and Assistance
- Database Health Checks

SAGE Computing Services Pty. Ltd. aims to provide the best Oracle technology experts to assist our clients in building, implementing and managing business solutions using Oracle software. Sage offers custom training workshops in Oracle products and a complete analysis, design, development and database administration consulting service.

SAGE is a small company which has succeeded by adopting a partnership approach with our clients. We have a long term relationship with many of our clients which is dependent on our providing services which achieve successful outcomes.

enquiries@sagecomputing.com.au

Course Descriptions

(not a full list of available courses, please see www.sagecomputing.com.au for a full course catalogue)

All courses can be customised to suit client requirements

enquiries@sagecomputing.com.au

Oracle JDeveloper 10.1.3 — 5 days

This course is designed to provide students with the skills and knowledge required to develop a web application using ADF Business Components and ADF Faces (JavaServer Faces) web pages within Oracle JDeveloper 10.1.3.

- JDeveloper Introduction
- JDeveloper IDE
- Database Connections
- Online Database Browser
- Stored Procedure Editor
- Workspaces and Projects
- Offline Database Modelling Tools
- Java Language
- Application Development Framework
- ADF Business Components Diagrammer
- ADF Business Components
- ADF Entity Objects
- ADF Associations
- ADF Domains
- ADF Application Modules
- ADF View Objects
- ADF View Links
- ADF Business Component Java Classes
- ADF Binding Layer and the ADF Model Layer Classes
- ADF Faces
- ADF Form
- ADF Read-Only Table
- ADF Input, Output and Command Components
- ADF Forms, Tables, Trees
- JSF Configuration, Navigation, Managed Beans and Event Handling
- Expression Language
- ADF Page Layout and Menus
- ADF Faces Skins
- Deploying an ADF Application
- Web Security
- ADF Selection, Validator, Converter and Visual Components
- Code Management and Source Control
- Advanced JDeveloper

Oracle 10G - New Features for Developers Workshop — 1 day

Aimed at developers, this course is designed to provide the student with an understanding of the new features of Oracle 10G.

- SQL*Plus Enhancements
- Flashback
- Enhanced SQL commands
- New SQL Features
- Data Pump
- PL/SQL Enhancements
- PL/SQL Packages
- Performance Enhancements

Application Express Workshop (previously HTMLDB) — 3 days

The course is designed to provide the student with the skills and knowledge required to develop a complete application using Oracle's Application Express product. The student will develop web interfaces (including forms, reports and charts; addition of validation and customised formatting) to create a small application.

- Product Overview
- SQL Workshop
- Utilities
- Application Builder
- Creating Pages
- Regions and Items
- Page Processing
- Shared Components
- Themes and Templates
- Other Page and Region Types
- Utilities and Reporting
- Advanced Development Techniques
- Administration and Deploying an Application

Oracle SQL and SQL*Plus Workshop - Oracle 10G Rel2 — 4 days

This course is designed to provide the student with a basis for developing systems using the Oracle database. The SQL language is covered from simple to complex constructs. Guidelines are provided on writing SQL for optimum performance and ease of maintenance.

- The Relational Model
- The SQL Language
- SQL*Plus and iSQL*Plus
- Oracle SQL Developer
- More about SELECT
- Substitution Variables
- Using SQL*Plus for Formatting Output
- Functions
- Joins
- Group and Analytical Functions
- Set Operators
- Subqueries
- Data Manipulation Language
- Database Objects
- Constraints
- Views and Sequences
- Indexes
- Clusters
- Security
- Locking and Read Consistency
- More Advanced SQL

PL/SQL Workshop — 3 days

This course is for developers who will be designing or building applications using the Oracle server. It is relevant for developers who are using the Oracle Developer toolset, and for those using alternative front-end products accessing the Oracle database. The course covers basic PL/SQL syntax and the use of server level procedures, functions and triggers.

- PL/SQL Overview
- Basic PL/SQL Syntax
- SQL Statements in PL/SQL
- Procedural Statements - Assignment and Conditional Processing
- Procedural Statements - LOOPS
- Exceptions
- Nested Blocks and Cursors
- Tables, Arrays and Records
- Architecture Overview
- Procedures / Functions
- Execution and Error Handling
- Security and Dependency
- Packages
- More About Packages
- Supplied Packages
- Triggers
- More About Triggers
- Large Objects

Oracle Forms Developer Workshop — 5 days

This course is designed for developers who will be designing or building applications using Oracle Form Builder. This is a practical course in which the student builds an application during a series of workshop sessions.

- Running a Form
- Forms Modules and Storage
- Working in the Builder
- Creating a Form
- Form and Data Block Properties
- Form Layout
- Items
- Introduction to Triggers
- Program Units
- Check Boxes, Radio Groups and List Items
- Other Item Types
- Visual Attributes
- Mouse Events
- Relations
- Alerts and Editors
- Lists of Values
- Record Groups
- Windows and Canvases
- Transaction Processing and Triggers
- Advanced Data Block Properties
- More Trigger Events
- Determining Form Properties
- Timers
- Integrating Multiple Forms Modules in an Application
- Forms Architecture and Java
- Integrating Forms with Reports
- PL/SQL Library Modules
- Managing Application Development
- Menu Modules

Oracle Reports Workshop - 10G — 4 days

This course is designed for developers who will be designing or building applications using Oracle Reports. This is a practical course in which the student builds a series of reports ranging from simple to complex.

- Product Overview
- The User Interface
- The Designer Interface
- Storage
- Building a Paper Report
- The Data Model Editor
- Other Query Types
- Columns
- Multiple Queries and Links
- The Paper Layout - Basic Objects
- Standard Layouts
- General Paper Layout Properties
- Advanced Paper Layout Properties
- Web Reports
- Displaying Files, Images and Charts
- Matrices
- Parameters
- PL/SQL in Reports
- Report Templates
- Publishing Reports on the Web

Oracle 10G – Database Administration Workshop — 5 days

This course is designed for Database Administrators. It covers the architecture of the Oracle 10g server, and the procedures required to effectively administrate the database. The course provides a series of practical workshops in which the students can practice the database administration techniques they have learnt.

- Oracle 10g Overview
- Oracle 10g Architecture
- Database Creation
- Startup and Shutdown and Oracle Database
- Oracle Enterprise Manager
- Database Structure
- Managing Tablespaces
- Managing Redo Log Groups and Members
- Database Storage
- Managing Undo
- Security
- Optimisation
- Database Tuning
- The Multi Threaded Server
- Backup and Recovery
- Data Pump

Application Tuning Workshop 10g — 3 days

This course is designed for Designers, Developers, and Database Administrators, and examines all aspects of tuning SQL statements and applications.

- Defining a Tuning Methodology
- Diagnostic Tools
- Processing an SQL Statement
- Indexes
- Cost Based Optimisation
- Gathering Statistics
- Stored Outlines
- Storage Parameters
- Hash Clusters and Index Clusters
- Optimising PL/SQL
- Optimising Applications through Stored Procedures and Packages
- Data Design for Performance
- Tuning Tips
- Partitions
- Optimise using Parallelisation
- Tuning Tools

Oracle Discoverer Workshop — 2 days

This course is designed for End Users and examines all aspects of using the latest versions of Oracle Discoverer. Both the web and client server interfaces are covered.

- Oracle Discoverer Overview
- Discoverer Workbooks
- Worksheets and Conditions
- Performing Analysis
- Customising Workbooks and Worksheets
- Printing and Exporting Query Results
- Scheduling and Administration

Oracle Portal Workshop — 3 days

This course is designed to provide the student with the knowledge and skills required to build corporate portals. The course covers the use of Oracle Portal for content management and includes recommendations and guidelines on the classification and searching of content. The standardisation and customisation of the Portal interface and styles are described. The workshop includes the use of Portal to create simple application components such as forms, reports and graphs. Finally the security management of a corporate portal is considered.

- Product Overview
- Page Groups and Pages
- Styles, Navigation Bars and Templates
- Item Regions and Classification
- Custom Types, Parameters and Events
- Application Components - Forms
- Application Components - Reports
- Shared Component
- Other Components
- Security

Advanced Oracle Portal Workshop — 1 day

The course is designed to provide the student with the knowledge and skills required to build custom portlets. The course describes the provider and portlet structure and the integration and management of custom portlets within the product. The attendee builds a simple provider and its custom portlets based on an example. The course focuses on PL/SQL portlets to demonstrate the techniques required, but also covers web portlets. Detailed coverage of the Portal API and its use, not only in custom portlets, but to enhance other Portal components is included.

- Programmable Portlets - Concepts
- PL/SQL Providers
- PL/SQL Portlets
- API Services – Session Storage
- API Services – Other Utilities
- Web Providers

Oracle End User Workshop — 2 days

This course is designed for End Users who require a knowledge of SQL to query the Oracle database. It commences with a description of relational concepts and continues with coverage of the SQL statements required to access information from one or more Oracle Tables. Some basic formatting is also covered.

- The Relational Model
- Structured Query Language
- SQL*Plus
- Oracle SQL Developer
- More About SELECT
- Substitution Variables
- Using SQL*Plus for Formatting Output
- Functions
- Joins
- Group Functions

All our training is conducted at the client site and using the client's Oracle licences. Sage Computing Services provides all course materials which the attendees retain after the course as a reference.

All Sage trainers are consultants who are using the products in real world situations and can bring a wealth of experience to the classroom.

The Java Language

Introduction

In this workshop we will be looking at the following:

Java Language
History
Object-Oriented Concepts
Implementing Java
Conditional Processing
Classes
Java Programming Tips
Other Java Language Concepts

Objective

To implement Java in the Oracle database, or to understand tools like JDeveloper and its frameworks, it is useful to have an understanding of the Java Language.

History

At its inception at Sun, Java was known as the *Stealth Project*, then the *Green Project* followed by *Oak*.

Finally it was released as Java in 1996 as v1.0. The following gives the version numbers and respective codenames:

Version	Year	Revision
1.0	1996	Initial revision.
1.1	1997	Updated
1.2	1998	Codename Playground. Re-branded as Java 2.
1.3	2000	Codename Kestrel.
1.4	2002	Codename Merlin.
5.0	2004	Codename Tiger.
6.0	2006	Codename Mustang.
7.0	tba	Codename Dolphin.

While these versions have added and extended the language, the Java libraries written to extend the functionality of the language have grown from a few hundred to around 3000 in v5.0.

In designing Java the Sun designers had 5 main goals:

- Use the Object-Oriented paradigm.
- Borrow parts from the older OO language C++ (and C).
- Platform independence.
- Execute remote sources securely.
- Built-in support for computer networks.

Object-Oriented Concepts

To understand the Java language, an understanding of the Object-Oriented (OO) paradigm is useful. Java is considered an Object-Oriented Programming (OOP) Language (OOPL), based around the key concepts of the *object*, *encapsulation, inheritance* and *polymorphism* among others*:*

- **Object** - an Object-Oriented program is one that is made up of a number of interacting objects.

  An object itself is a conceptual entity, such as a person, account or queue. The main defining aspects of an object are its:

  - **Name** – identifies an object from all other objects.
  - **Attributes** – synonymously referred to as data members, variables and fields, these hold the data representing the object. These in turn hold the state of the object.
  - **Methods** – that access and operate on the data members in a predefined manner, essentially defining an object's behaviour, and its interface to other objects.

- **Class** - an object is a single *instance* of a *class*. A class is the template mechanism for object instances defining the attributes and methods at runtime.

  The act of creating an object *instance* of a class is called the *instantiation* of that class.

- **Messages** - objects talk to each other through *messages*. In a technical sense this is implemented by one object's method invoking another object's exposed method. The objects are referred to as the *caller* and *receiver* respectively, or alternatively both may be referred to as *agents.*

  In constructing the message a number of arguments may be optionally passed.

  In response to a message the receiver will carry out a number of actions as defined by the invoked method.

- **Associations** – a class may have an *association* with another class. The association describes a relationship between the two classes. For example a person object may "subscribe-to" a magazine object, or a queue object may "contain" booking objects.

  The association also describes the *multiplicity* or *cardinality* of the relationship. For example the queue may contain 0, 1 or many bookings objects. Such multiplicity is typically described in terms of : 0..1, 1..1, 1..* or 0..* where * implies multiple.

- **Abstraction** – the ability for a program to ignore the implementation details of an object it is manipulating, giving it the ability to focus on the essential. In an OO system this is implemented through the concepts of encapsulation, inheritance and polymorphism.

- **Encapsulation** – a key concept behind Object-Oriented programming is the concept of *encapsulation*. Attributes and methods of an object are made either private to the class such that only its attributes and methods may see the private constructs, or alternatively public such that not only the class can see these constructs, but outside objects as well. Such publicly exposed attributes and methods become the *interface* of which outside objects may access and manipulate the object.

  The key reason for encapsulation is to control the access to the internal attributes and methods, essentially sealing the class's data safely inside the class. As such the internal data may only be accessed and modified through the class's own trusted public methods.

- **Inheritance** - classes can be organised into a hierarchical *inheritance* structure, where a child class inherits attributes from a parent class. Also known synonymously as an inheritance tree.

  A child class is synonymous with the term *subclass* or *derived class*. The parent class is synonymous with the terms *superclass* or *base class*.

  The inheritance tree may run many levels deep. For example a person is derived from mammal class, also derived from an animal superclass.

  In a pure Object-Oriented system, a class may support *multiple inheritance.* For example a person may inherit from both the student and employee superclasses.

- **Polymorphism** - one object calls another through a message, specifying a called method. If that method is found within the receiver class, the method is invoked. If the method is not found the receiver class's parent class equivalent method is invoked, and so on. This behaviour is called *polymorphism* - derived from the Greek terms meaning *many-forms.*

  Polymorphism extends not only to the methods but also the attributes of an object.

Note however Java is not considered a pure Object-Oriented programming language because of the following limitations:

- Support for primitive datatypes
- Single inheritance
- No support for operator overloading

Implementing Java

The basics of implementing a Java program are as follows:

Source Code, Bytecode and the JVM

Java source code is stored in a human readable text file with a `*.java` extension. In order to provide platform independence, the Java compiler converts syntactically correct source code into *bytecode*. Bytecode is stored in a file with a `*.class` extension. Each bytecode file has a 1-to-1 relationship with the relating `*.java` file.

The bytecode is platform independent and can be transferred between platforms. The bytecode is run on a platform specific Java Virtual Machine (JVM) that interprets the bytecode and runs the equivalent platform specific command.

The ultimate goal of this approach is “Write Once, Run Anywhere”.

JRE and JDK

The Java Runtime Environment (JRE) is the software required to run a Java program. You can obtain JRE releases from the Sun website.

The core components of the JRE are:

- Precompiled bytecode Java libraries for your programs to execute. This includes core libraries such as lists, security and internationalisation, integration libraries such as JDBC and JNDI, and user interface libraries such as Swing and AWT.
- A platform independent Java Virtual Machine (JVM) to run compiled bytecode.

The Java Development Kit (JDK), sometimes known as the Software Development Kit (SDK), is a superset of the Java Runtime Environment (JRE). It includes additional tools for development including a Java compiler, documentation and a debugger. JDeveloper includes a version of the JDK in order to run your programs.

J2SE, J2EE and J2ME

The Java Development Kit (JDK) comes in three flavours, namely:

- **J2SE** - specifically Java 2 Standard Edition (or Java SE) is essentially a set of Java APIs useful to any Java program. J2SE is contained both within J2EE and J2ME. This includes libraries for maths, IO, text, as well as graphical toolkit libraries such as Swing.
- **J2EE** - specifically Java 2 Enterprise Edition (or Java EE) is designed to run distributed multi-tier applications. J2EE includes a number of Java APIs to support enterprise development including EJBs, JSPs, and web service technologies.
- **J2ME** - specifically Java 2 Micro Edition is a set of Java APIs aimed at the embedded processor market included within PDAs and mobile phones for example.

Class

An example of Java class is as follows:

```
01  package au.com.sagecomputing.cars;
02
03  import java.lang.*;
04
05  /*
06     This class defines a Bmw
07  */
08  public class Bmw extends Cars {
09    int passengers;  // override the Cars passengers
10    int kmsPerLitre; // unique to Bmw
11
12    public Bmw() {
13      passengers = 4;
14      maxKph = 250;   // maxKph is inherited from Cars
15      kmsPerLitre = 9;
16    }
17
18    // super.getMaxKph is the same as getMaxKph() here
19    public int getHourlyFuelUsage() {
20      return super.getMaxKph() / kmsPerLitre;
21    }
22
23    public int getPassengers() {
24      return passengers;
25    }
26
27    public void setPassengers(int newPassengers) {
28      passengers = newPassengers;
29    }
30  }
```

This class definition identifies a number of concepts:

- **Packages** - each Java class resides within a package. A package typically contains related classes. The package name is defined with a dot notation.

- **Import** – a Java class may import zero, one or many classes to make use of the code supplied within. As the other class is stored in a package, it is referenced with dot notation. If more than one class is required a wildcard may be used.

- **Comments** – comments may be limited to one line and preceded with two forward slashes `//` or stretched over multiple lines enclosed in slash-star notation such as `/* comment */`.

- **Class Declaration** – a class typically has a name. It can *extend* or *inherit* from a parent class. Defining the class as *public* exposes the class to classes outside of its package.

- **Variables** – or instance variables are the internal attributes of the class. Typically an attribute has a datatype and name. Implicitly, if not defined the datatype is considered *private* to the class. In other words only attributes and methods within this class can see and modify the variable.

- **Constructors** – a class may have zero, one or many constructors. A constructor is a type of class method that is intended to be called when an instance of the class is instantiated. Constructors typically initialise the object ready for first use. Unlike other methods they do not return a value.

  Constructors, like other constructs in the class may be defined as *public* or *private* or *protected.* A protected construct is one that is accessible to the current object and any other classes in the same package, but not to classes external to the package.

  In your code, to instantiate an object instance using a constructor you may use code like the following:

```
01  Bwm myBwm = new Bmw();
```

- **Methods** –in addition to the constructor, the class may also include a number of methods. Methods define a return datatype or `void` if they do not return a value. It is expected that a method that returns a datatype includes a return statement.

  Methods may accept parameters and arguments. In the same manner as other class constructs the method may expose itself through the public, private or protected keywords.

- **Scope** – attributes and arguments have scope, defining where they can be used within a class. An attribute defined in the class is accessible to the rest of the class. However a variable or argument defined within a method is only accessible to that method.

- **Class (or Static) Attribute** – is one that only exists once for all object instances. It's defined with the `static` keyword. It is useful for aggregates, for example:

```
01  private static int numberPatients = 0;
```

- **Class (or Static) Method** – similar to a class attribute, is a method that exists for the class, not each individual object instance. The keyword `static` is also used. A static method may be accessed in two ways. Say we have an additional static method in our Bmw class:

```
01  public static void writeMessage() {
02    System.out.println("Hello JDeveloper!");
03  }
04
05  Cars.writeMessage();
06  // or
07  Cars cars = new Cars();
08  cars.writeMessage();
```

- **Constants** – an attribute whose value cannot be changed may be defined as a constant, using the `final` keyword. As constants are good candidates to be only declared once for the entire class, they often use the static keyword:

```
01  private static final int maxPatients = 0;
```

- **Code Blocks** – code is enclosed in curly brackets `{ }`. For example the components of a class are enclosed in brackets, so too is the code for a constructor or a method.

- **Semicolons** – individual operations, such as importing a package, defining a class attribute or assigning a value to another variable, are terminated with a semicolon. Operations may be split over several lines completed with a semicolon. Typically declarations and associated code blocks, such as a method or class definition do not end in a semicolon.

- **Case Sensitive** – Java is a case sensitive language. If you define a variable as `maxBookings`, you cannot refer to it as `MaxBookings` or `MAXBOOKINGS`. Only `maxBookings` is valid. This is true for all Java constructs, including reserved words. Reserved words are always in lowercase.

Naming Conventions

Java does not enforce specific naming conventions. However an industry standard and guidelines by Sun have been developed in helping Java developers write standard code.

Default Names

Given the case sensitive nature of the Java language, there is a default naming convention for language's constructs:

- Names are alphabetic.

- Underscores and punctuation are not used.

- Packages – use a dot naming convention, usually containing as the first part the reverse URL of the organisation, followed by component package descriptions. For example for a queue mechanism written by Sage Computing Services, we might use `au.com.sagecomputing.collections.queue`.

- Class – starts with a capital letter, and each subsequent word starts with a capital letter. eg. `EmployeeQueue` or `ConnectionPoolFactory`

- Class Attributes – start with a lowercase letter, and each subsequent words starts with a capital, for example `maxWages` or `currentStateValid`.

- Class Methods – similar to class attributes, start with a lowercase letter, and each subsequent words starts with a capital, for example: `isObjectValid()` or `getMaxWages()`.

Reserved Words

The Java Language Specification defines a number of keywords:

```
http://java.sun.com/docs/books/jls/third_edition/html/lexical.html
```

```
abstract        default         goto            package         this
assert          do              if              private         throw
boolean         double          implements      protected       throws
break           else            import          public          transient
byte            enum            instanceof      return          true
case            extends         int             short           try
catch           false           interface       static          void
char            final           long            strictfp        volatile
class           finally         native          super           while
const           float           new             switch
continue        for             null            synchronized
```

These keywords are reserved and may not be used for identifier names in your Java programs.

Data Types

Central to learning Java is the concept of primitive data types and those based on Classes. A key area of difficulty for beginner Java programmers is converting between the primitive data types and classes.

Primitive Data Types

Java supports a number of primitive data types such as boolean and int.

Type	Size	Default	Contains	Example	Range
`boolean`	N/A	`false`	true or false	`true`	
`char`	16 bits	`'/u0000'`	Unicode character	`'q'`	`\u0000 to \uFFFF`
`byte`	8 bits	`0`	Signed integer	`4`	`-128 to 127`
`short`	16 bits	`0`	Signed integer	`92`	`-32768 to 32767`
`int`	32 bits	`0`	Signed integer	`331`	`-2147483648 to 2147483647`
`long`	64 bits	`0`	Signed integer	`924321`	`-9223372036854775808 to 9223372036854775807`
`float`	32 bits	`0.0`	Floating point	`453.24`	`1.4E-45 to 3.4028235E+38`
`double`	64 bits	`0.0`	Floating point	`924521.421`	`4.9E-324 to 1.7976931348623157E+30`

Further comments:

- **Boolean** – supports two possible values of true or false and can represent the truth of an expression, yes or no, or on or off.
- **Char** – a 16 bit type representing a Unicode character.
- **Byte, Short, Int** and **Long** – similar integer types only varying in the range of numbers they can store. Includes positive and negative numbers.
- **Float** and **Double** – similar floating point types only varying in the range of the numbers they can store. Includes positive and negative numbers, as well as a decimal component.

Java allows conversion between the integer and floating types. As a char may also represent a number it can also be converted between the types. The only restricted data type is the boolean which cannot be converted.

Strings

Strings are implemented in `java.lang.String`. Internally it stores an array of Unicode characters. The array is only accessible through the String class's API.

To create a String:

```
01 String myString = "Hello " + " Java";
```

The String class provides a number of methods including:

```
01 String myNumber  = String.valueOf(15);   // converts int to String
02 String myFloat   = String.valueOf(32.1); // converts float to String
03 String myBoolean = String.valueOf(true); // converts boolean to String
04 String myString  = "TRUE";
05 String abc       = "abc";
06 String def       = "def";
07
08 System.out.println(myNumber.equals(myFloat));             // false
09 System.out.println(myBoolean.equalsIgnoreCase(myString)); // true
10
11 System.out.println(abc.compareTo(def));                   // -3
12
13 System.out.println(abc.startsWith(def));                  // false
14 System.out.println(abc.startsWith("ab"));                 // true
15
16 System.out.println(abc.indexOf("b"));                     //  1
17 System.out.println(abc.indexOf("x"));                     // -1
18
19 System.out.println("  test  ".trim());                   // "test"
20 System.out.println("test".toUpperCase());                // "TEST"
21 System.out.println("TEST".toLowerCase());                // "test"
22
23 System.out.println("abcdefghijklmn".substring(4, 8));    // "efgh"
24 System.out.println("cat-dog".replace("-", " & "));       // "cat & dog"
25
```

Dates

Dates and times may be stored in Java as `long` values, or via `java.util.Date` or `java.util.Calendar`.

Behind the scenes Java stores all date and time values as a `long` representing the number of milliseconds since midnight January 1st 1970 measured from Universal Time (UTC), known as the *epoch*. The following statement retrieves the current date-time in milliseconds from the epoch:

```
01 long now = System.currentTimeMillis();
```

The `java.util.Date` class is a wrapper on the long approach, and supplies methods to:

```
01 long now  = System.currentTimeMillis();
02 long then = System.currentTimeMillis() - (1000 * 60 * 60 *24);
03
04 Date nowDate = new Date(now);
05 Date thenDate = new Date(then);
06
07 System.out.println(nowDate.toString());          // Prints current date
08 System.out.println(thenDate.toString());         // Prints then date
09 System.out.println(nowDate.equals(thenDate));    // false
10 System.out.println(nowDate.compareTo(thenDate)); // 1
11 System.out.println(nowDate.before(thenDate));    // false
12 System.out.println(nowDate.after(thenDate));     // true
```

Arrays

Java allows you to define fixed length arrays of elements in two fashions:

```
01 String[] stringArray = new String[3];
02 String[] stringArray = new String[] { "Red", "Green", "Blue" };
```

Array elements are accessed in the following manner:

```
01 String[] stringArray = new stringArray[] = { "Red", "Green", "Blue" };
02
03 for (int i = 0; i < stringArray.length; i++)
04   System.out.println(stringArray[i]);   // prints Red, Green, Blue
05
06 stringArray[1] = "Purple";
```

Accessing an element outside of an array's bounds would raise an `ArrayIndexOutOfBoundsException` exception.

Arrays are designed to store both primitive datatypes and object references.

Collections

Beyond the simple array structure, Java supports collections. Collections are designed to support a collection of *objects* (eg. instances of classes), where the number of objects may vary dynamically during the life of the program.

While the Collection classes are powerful, they cannot store primitive datatypes. To store a primitive datatype you must use a *Boxing Primitive* class (see next).

Collections are implemented as classes themselves in the Java Collections Framework. The Framework defines a number of interfaces such as:

- `List` – navigable collection of objects, eg. `ArrayList`
- `Set` – a collection with no duplicate elements, eg. `HashSet`
- `Map` – a collection of key-value pairs, eg. `HashMap`

An example usage of `ArrayList`:

```
01 ArrayList students = new ArrayList();
02
03 students.add(new Student("Chris", "Muir"));// append student end of list
04 students.add(0, new Student("J", "Doe"));  // add student start of list
05 Student student = students.get(0);         // retrieves J Doe
06 System.out.println(students.size());       // prints 2
07 students.remove(student);                  // removes J Doe from list
08 students.clear();                          // clears entire list
```

The full list of classes that implement the Collections Framework interfaces are located in `java.util`. Consult the Java documentation for more information on usages.

Boxing Primitives

A large amount of classes, including the Collection classes cannot store primitive datatypes, only objects. In order to store primitive data types you must use one of the boxing classes:

Primitive	Boxing class
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`integer`	`Integer`
`short`	`Short`
`long`	`Long`
`float`	`Float`
`double`	`Double`

The bloxing class is a fully fledged Java class that can be instantiated with a primitive datatype, to *wrap* the primitive such that it can be used and stored within an object framework. eg:

```
01 import java.util.ArrayList;
02
03 ...
04
05 int myInt1 = 5;
06 int myInt2 = 7;
07
08 ArrayList results = new ArrayList();
09
10 Integer result1 = new Integer(myInt1);
11 Integer result2 = new Integer(myInt2);
12
13 results.add(result1);
14 results.add(result2);
```

To convert between a primitive numeric type and the String class, you must make use of the intermediate boxing primitive class. For example to convert from a String to a float, or vice versa, the Float boxing primitive class must be used:

```
01 float myFloat = Float.valueOf("1234");      // String to Float to float
02
03 String myString = Float.toString(myFloat);  // float to Float to String
```

main() method

The `main()` method is a special method which we can add only once to a single class within our application.

When attempting to run a program, Java searches for `main()` to start your program. In other words `main()` is the main entry into your program provinding the starting point.

You define `main()` as follows:

```
01 public class SomeClass {
02
03   public static void main(String[] args) {
04     for (int i = 0; i < args.length; i++)
05       System.out.println(args[i]);
06   }
07 }
```

`main()` must be defined as a `public` `static` `void` method. It accepts an array of Strings from the command line named `args`. Running the above code with the following command line results in:

```
01 java SomeClass "hello" "Java" "Developers"
02
03 hello
04 Java
05 Developers
```

Operators

Java supports a number of operators that may be used in expressions and assignments:

Comparison Operators

>	Greater than	x > y	Returns true when x is greater than y
<	Less than	x < y	Returns true when x is less than y
>=	Greater than or equal to	x >= y	Returns true when x is greater than or equal to y
<=	Less than or equal to	x <= y	Returns true when x is less than or equal to y
==	Equality	x == y	Returns true when x is equal to y
!=	Inequality	x != y	Returns true when x is not equal to y

Boolean Operators

||	Or	x || y	Returns true when either x or y is true
&&	And	x && y	Returns true when both x and y are true
!	Not	!x	Returns true when x is false
^	Different	x^y	Returns true when x and y are different

Assignment Operators

=	Assign	x = y	Assign y to x
+=	Add variables	x += y	Add y to x and assign to x
-=	Subtract variables	x -= y	Subtract y from x and assign to x
*=	Multiple variables	x *= y	Multiple y by x and assign to x
/=	Divide variables	x /= y	Divide x by y and assign to x
%=	Modulus of variables	x %= y	Find the modulus of y into x and assign to x

Arithmetic Operators

+	Addition	x + y	eg. 5 + 4 = 9
-	Subtraction	x - y	eg. 5 - 4 = 1
/	Division	x / y	eg. 6 / 3 = 2
%	Modulus	x % y	eg. 7 % 3 = 1

Increment/Decrement Operators

++	Increment	++x or x++
--	Decrement	--y or y--

Bitwise and Shift Operators

~	Unary of Complement	~x	Inverts data type bites
&	And	x & y	Bitwise and on 2 values' bits
!	Or	x | y	Bitwise or on 2 values' bits
^	Exclusive Or	x ^ y	Bitwise Xor on 2 values' bits
<<	Left shift	x << 2	Shift bits to the left by 2
>>	Right shift	x >> 3	Shift bits to the right by 3

Brackets may be used in changing the outcome of an expression and its operators.

Expressions using operators evaluate in a left to right order in order of brackets, indices, multiplication, division, addition and subtraction operators first. For more information regards the evaluation order, refer to the Java specification:

http://java.sun.com/docs/books/jls/third_edition/html/expressions.html

The ternary operator requires its own explanation. The ternary operator is a short hand form for writing if statements. For example:

```
01 int myValue = (x && y) ? 4 : 6;
```

It reads if `x && y` are `true`, then return `4`, else `6` to `myValue`. The expression before the ? is the boolean expression to be evaluated. The first value after the ? is the result to be returned if the boolean expression evaluates to true, else the second value after the `:` is returned.

Conditional Processing

Similar to structured languages, Java includes the ability to undertake condition processing:

If statement (and if else)

Java supports the following `if` constructs:

```
01 if (x < y)
02   doSomething();
03
04 if (x < y) {
05   doSomething();
06   doSomethingAgain();
07 }
08
09 if (x < y) {
10   doSomething();
11 } else {
12   doSomethingElse();
13 }
14
15 if (x < y) {
16   doSomething();
17 } else if (x > y) {
18   doSomethingIfElse();
19 } else {
20   doSomethingElse();
21 }
```

Switch

Java supports the `switch` construct:

```
01 int colour = 2;
02 switch (colour) {
03   case 1  : System.out.println("Colour is Blue");
04             break;
05   case 2  : System.out.println("Colour is Green");
06             break;
07   case 3  : System.out.println("Colour is Red");
08             break;
09   default : System.out.println("Colour is most likely Brown");
10             break;
11 }
```

while loop

Java supports the following `while` loop construct:

```
01  int i = 0;
02  while (i < 10) {
03    System.out.println(i);
04    i++;
05  }
```

do loop

Java supports the following `do` loop construct:

```
01  int i = 0;
02  do {
03    System.out.println(i);
04    i++;
05  } while (i < 10);
```

for loop

Java supports the following `for` loop constructs:

```
01  for (int i = 0; i < 10; i++)
02    System.out.println("Number " + i);
03
04  for (int i = 0, j = 10; i < j; i++, j--)
05    System.out.println("Number " + i);
06
07  for (int i = 0; i < 10; i++) {
08    System.out.println("Number " + i);
09    System.out.println("umm..... ");
10  }
```

As of Java 5.0 the following `for` loop construct is also supported, synonymously known as the for/in or for-each loop:

```
01  String[] colours = new String[] {"Red", "Green", "Blue"};
02  for (String s : colours)
03    System.out.println(s);
```

Exceptions

All exceptions raised in Java are classes. There are two exception base classes `java.lang.Error` and `java.lang.Exception`. The `Error` class is reserved for critical errors such as out of memory. The `Exception` class is for programmatic exceptions raised by the programmer. Typically other exception classes will extend these two base classes.

Java supports exception handling with the `try` & `catch` & `finally` constructs:

```
01  try {
02    doSomething();
03  }
04  catch (AnException e1) {
05    doSomethingE1();
06  }
07  catch (AnotherException e2) {
08    doSomethingE2();
09  }
10  finally {
11    doSomethingFinally();
12  }
```

Exceptions can only be handled within the `try` & `catch` block. Exceptions occurring outside this code will not be handled unless higher on the stack a calling module has surrounded the block with `try` & `catch`.

The catch block must explicilty state the Exception class it is handling. As all Exception subclasses either extend the `Exception` or `Error` superclasses, to write a generic catch handler you need only specify either of these classes. Otherwise specify the specific `Exception` subclass.

The `finally` clause is optional. It is guaranteed to be called if 1 or more lines are executed in the `try` block. It is ideal location for code to clean up any work done in the `try` block such as closing open file handlers.

Classes

Beyond defining a simple class, Java also supports the following class constructs:

Object Instantiation and By-Reference Variables

To instantiate an object, you must first declare a by-reference variable. Multiple by-reference variables may act as a *handle* to the same instantiated object.

```
01  Car firstCar = new Car("BMW");
02  Car secondCar;
03
04  if (secondCar == null)
05    System.out.println("secondCar is null"); // secondCar is null
06
07  secondCar = firstCar;
08
09  if (secondCar != null)
10    System.out.println(secondCar.getName()); // BMW
11
12  if (firstCar == secondCar)
13    System.out.println("Only 1 car"); // Only 1 car
14
15  firstCar = null;
16
17  String carName = firstCar.getName(); // Raises null pointer exception
```

Extends

Class inheritance is implemented through the `extend` keyword:

```
01  public class Person {
02    String surname = "Muir";
03    String firstName = "Christopher";
04
05    public String getName() {
06      return surname;
07    }
08  }
09
10  public class Student extends Person {
11    String studentNumber = "44323";
12    String firstName = "Chris";
13
14    public String getName() {
15      return surname + " " + firstName; // returns "Muir Chris"
16    }
17
18    public String getSurname() {
19      return super.surname;             // returns "Muir"
20    }
21
22    public String getFirstName() {
23      return this.firstName;            // returns "Chris"
24    }
25  }
```

Abstract and Concrete Classes

An abstract class is one that is not instantiated directly, but rather is a template defining that its subclasses must implement certain methods:

```
01  public abstract class Person {
02    String surname = "Muir";
03    String firstName = "Christopher";
04
05    abstract String getName();
06  }
07
08  public class Student extends Person {
09    public String getName() {
10      return surname + " " + firstName;
11    }
12  }
```

A concrete class is a class that extends an abstract class and all the abstract methods required by the abstract class.

Interfaces

Similar to abstract classes, you may also define an interface class. An interface class is not instantiated directly, but rather is a template defining that its subclass must implement certain attributes and methods:

```
01  public interface BankAccount {
02    final int MIN_VALUE = 0;
03    final double INTEREST_RATE = 1.004;
04    double getBalance();
05    void applyInterestRate();
06  }
07
08  public class SavingsAccount implements BankAccount {
09    double balance = 50;
10
11    public double getBalance() {
12      return balance;
13    }
14
15    public void applyInterestRate() {
16      balance *= INTEREST_RATE;
17    }
18  }
```

The difference between an abstract and interface class, is an abstract class allows you to define abstract methods that are fully implemented methods. Methods defined in an interface class are totally abstract; in other words an interface method can not have any implementation code.

Nested Classes

You may define nested classes:

```
01  public class Parent {
02    public class Child {....}
03  }
```

The Child class is only accessible by the Parent class via the Child class's public and protected methods. All contents of the Parent class are accessible to the Child.

Anonymous Classes

You may declare unnamed classes that implement interfaces. For example:

```
01  account.registerNewBankAccount(new BankAccount() {
02    double balance = 50;
03
04    public double getBalance() {
05      return balance;
06    }
07
08    public void applyInterestRate() {
09      balance *= INTEREST_RATE;
10    }
11  });
```

This creates an anonymous instance of the `BankAccount` interface class with the required methods `getBalance` and `applyInterestRate` and a local attribute balance. The `account.registerNewBankAccount` method takes the anonymous class as a parameter.

Java Programming Tips

The following are some Java programming tips for programmers starting out with the Java language, hopefully assisting in avoiding common pitfalls:

Comparing object with == and equals()

The equality of variables can be tested in two fashions in Java, using the == equality operator or the object `equals()` method.

The equality operator is only effective for primitive data types. For example:

```
01 int myInt1 = 1;
02 int myInt2 = 1;
03
04 System.out.println(myInt1 == myInt2); // Returns true
05 myInt2 = 2;
06 System.out.println(myInt1 == myInt2); // Returns false
```

The equality operator cannot be used when comparing object types. Objects can be compared for equality through their implementation of the `equals()` method. For example:

```
01 Integer integer1 = new Integer(1);
02 Integer integer2 = new Integer(1);
03
04 System.out.println(integer1 == integer2);      // Returns false
05 System.out.println(integer1.equals(integer2)); // Returns true
```

If you create custom classes and wish to test them for equality, you need to define your own `equals()` method, programmatically undertaking the comparison and returning a `boolean` result.

As the `String` data type is an object, it also supplies an `equals()` method. To test the equality of 2 strings you should not use the == equality operator.

```
01 String apple = "Apple";
02
03 if (apple.equals("Orange")) ....
```

null

The `null` keyword is a special literal value meaning “no object”. The `null` value is unique in that it exists in all classes. It however does not exist for primitive types.

You may assign and test for `null` in your code, for example:

```
01 String myString = null;
02
03 if (myString == null)
04   System.out.println("true"); // prints true
05
06 if (null == null)
07   System.out.println("true"); // prints true
```

Note that in Java an empty string is not equivalent to null:

```
01 String myString = "";
02
03 if (myString == null)
04   System.out.println("true");
05 else
06   System.out.println("false"); // prints false
```

Other Java Language Concepts

In understanding Java there is key additional terminology which it is useful to be familiar with when reading Java documentation:

Design Patterns

Within software design, design patterns are standard solutions to common problems in software design. The concept of design patterns originated from architecture in the 1700s.

A design pattern itself is not a finished design. Rather it is a template for a solution to a problem that frequently occurs in designing computing systems. Within an Object-Oriented system it maps the relationships and interactions between classes and objects.

The concept does not extend to algorithms as they solve computational problems rather than design issues.

An understanding of Java does not need a detailed knowledge of design patterns. However they are frequently acknowledged in Java and Object-Oriented documentation, and as such an understanding of the concepts is useful.

Object Classification

In describing objects with certain characteristics, Java and Object-Oriented programmers often classify objects into the following types:

- **Singleton** – limited to a single existence in a program regardless of the number of users.
- **Mutable** – state can change at runtime.
- **Immutable** – instantiated with a fixed runtime state that does not vary.
- **First-class** - an object that can be used without restriction, typically at the finest level of object granularity (ie. String class)
- **Container** – containing other object.
- **Factory** – creates other objects

JavaBeans

A JavaBean is a class that is designed for reuse, to be stored in a container, and which conforms to a clear specification on how other Java code may interact with it.

A JavaBean does not have to implement or extend any other class to be a JavaBean. It is not however restricted from implementing or extending other classes.

At its simplest, a JavaBean will conform to the following rules:

- It must have a no-parameter constructor such that it may be created and manipulated by bean tools.

- It exposes internal attributes (called bean *properties*) to the outside via *accessor* routines, namely getter and setter methods. For example if the bean has a property `xValue`, then it should have a getter method `getXValue()` and setter method `setXValue()`.

- The exception is for boolean attributes where the getter routine can either have the prefix `get` or `is`, eg. `getMyBoolean()` or `isMyBoolean()`.

```
01 public class myBean {
02   int xValue;
03   int yValue;
04
05   myBean() {
06     xValue = 0;
07     yValue = 0;
08   }
09
10   public int getXValue() { return xValue; }
11   public int getYValue() { return yValue; }
12
13   public void setXValue(int newValue) { xValue = newValue; }
14   public void setYValue(int newValue) { yValue = newValue; }
15 }
```

The complete JavaBean specification can be found at:

`http://java.sun.com/products/javabeans/reference/api/index.html`

JavaBeans make ideal classes for UI components, where the user may interact with the JavaBean's properties via the accessor methods.

Garbage Collection

The Java *garbage collector* takes care of deallocating memory rather than the programmer having to explicitly do this. This is a key difference for programmers with, for example, C++ and C backgrounds. In C++ routines are supplied to allocate and deallocate memory from the heap when instantiating and destroying objects – `malloc()` and `free()` respectively. The main issue with this approach ist if the programmer forgets to free memory after use, a memory leak occurs consuming memory until the entire program is killed.

The garbage collector in Java takes care of deallocating memory for the programmer, meaning the programmer does not explicitly have to deallocate instantiated objects and thus avoids potential memory leaks. Once a program de-references an instantiated objects (ie. the objects falls out of scope of the current routine), the routine is marked ready for collection. At a periodic interval or certain event, the garbage collector is called and deallocates de-referenced objects such that the memory is reclaimed.