**Is java compiled or interpreted?**

Java uses a two step compilation process. Java source code is compiled down to "bytecode" by the Java compiler. The bytecode is executed by Java Virtual Machine (JVM). The current version of Sun HotSpot JVM uses a technique called [Just-in-time (JIT) compilation](http://en.wikipedia.org/wiki/Just-in-time_compilation) to compile the bytecode to the native instructions understood by the CPU on the fly at run time.

Some implementations of JVM might interpret the bytecode instead of JIT compiling it to machine code and running it directly. While this is still considered an "interpreter." It's significantly different from interpreters that read and execute the high level source code (i.e. in this case, Java source code is not interpreted directly, the bytecode, output of Java compiler, is.)

To summarize, depending on the execution environment, bytecode can be:

* compiled ahead of time and executed as native code (similar to C++)
* compiled just-in-time and executed
* interpreted
* directly executed by a supported processor (bytecode is the native instruction set of some CPUs)

**Can you mark a class as both abstract and final?**

You can't mark a class as both abstract and final. They have nearly opposite

meanings. An abstract class must be subclassed, whereas a final class must not be

subclassed. If you see this combination of abstract and final modifiers, used for a

class or method declaration, the code will not compile.

if even a single method is abstract, the whole class must be

declared abstract.

You can, however, put nonabstract methods in an abstract class.

**Difference between interface and abstract class?**

Think of an interface as a 100-percent abstract class.  
But while an abstract class can define both abstract and non-abstract methods, an interface can have only abstract methods.  
All interface methods are implicitly public and abstract. In other words,you do not need to actually type the public or abstract modifiers in the method declaration, but the method is still always public and abstract.  
All variables defined in an interface must be public, static, and final—in other words, interfaces can declare only constants, not instance variables.

Interface methods must not be static.

■Because interface methods are abstract, they cannot be marked final,

strictfp, or native. (More on these modifiers later.)

■**An interface can *extend* one or more other interfaces.**

■An interface cannot extend anything but another interface.

■An interface cannot implement another interface or class.

■An interface must be declared with the keyword interface.

■Interface types can be used polymorphically

**Object’s equal() method and when do you override it?**

if you don't override a class's equals()

method, you won't be able to use those objects as a key in a hashtable and you

probably won't get accurate Sets, such that there are no conceptual duplicates.

boolean equals (Object obj) Decides whether two objects are meaningfully equivalent.

int hashCode() Returns a hashcode int value for an object, so that the object can

be used in Collection classes that use hashing, including Hashtable,

HashMap, and HashSet.

**final**

final can be used to mark a variable "unchangeable"

private final String name = "foo"; //the reference name can never change

final can also make a method not "overrideable"

public final String toString() { return "NULL"; }

final can also make a class not "inheritable". i.e. the class can not be subclasses.

public final class finalClass {...}

public class classNotAllowed extends finalClass {...} // Not allowed

**finally**

finally is used in a try/catch statement to [execute code "always"](http://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html)

lock.lock();

try {

//do stuff

} catch (SomeException se) {

//handle se

} finally {

lock.unlock(); //always executed, even if Exception or Error or se

}

Java 7 has a [new try with resources statement](http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html) that you can use to automatically close resources that explicitly or implicitly implement [java.io.Closeable](http://docs.oracle.com/javase/7/docs/api/java/io/Closeable.html) or [java.lang.AutoCloseable](http://docs.oracle.com/javase/7/docs/api/java/lang/AutoCloseable.html)

**finalize**

finalize is called when an object is garbage collected. You rarely need to override it. An example:

public void finalize() {

//free resources (e.g. unallocate memory)

super.finalize();

}

# What Is the Static Keyword in Java?

By [**Doug Lowe**](http://www.dummies.com/search.html?query=Doug+Lowe) from [**Java For Dummies Quick Reference**](http://www.dummies.com/store/product/Java-For-Dummies-Quick-Reference.productCd-1118168232.html)

In Java, a *static* member is a member of a class that isn’t associated with an instance of a class. Instead, the member belongs to the class itself. As a result, you can access the static member without first creating a class instance.

The two types of static members are static fields and static methods:

* **Static field:**A field that’s declared with the static keyword, like this:

private static int ballCount;

The position of the static keyword is interchangeable with the positions of the *visibility keywords* (private and public, as well as protected). As a result, the following statement works, too:

static private int ballCount;

As a convention, most programmers tend to put the visibility keyword first.

The value of a static field is the same across all instances of the class. In other words, if a class has a static field named CompanyName, all objects created from the class will have the same value for CompanyName.

Static fields are created and initialized when the class is first loaded. That happens when a static member of the class is referred to or when an instance of the class is created, whichever comes first.

* **Static method:**A method declared with the static keyword. Like static fields, static methods are associated with the class itself, not with any particular object created from the class. As a result, you don’t have to create an object from a class before you can use static methods defined by the class.

The best-known static method is main, which is called by the Java runtime to start an application. The main method must be static, which means that applications run in a static context by default.

One of the basic rules of working with static methods is that you can’t access a nonstatic method or field from a static method because the static method doesn’t have an instance of the class to use to reference instance methods or fields.

**Transient Variables vs Volatile Variables**

**Transient Variables**

If you mark an instance variable as transient, you're telling the JVM to skip

(ignore) this variable when you attempt to serialize the object containing it.

Serialization is one of the coolest features of Java; it lets you save (sometimes called

"flatten") an object by writing its state (in other words, the value of its instance

variables) to a special type of I/O stream. With serialization you can save an object

to a file, or even ship it over a wire for reinflating (deserializing) at the other end, in

another JVM. Serialization has been added to the exam as of Java 5, and we'll cover

it in great detail in Chapter 6.

**Volatile Variables**

The volatile modifier tells the JVM that a thread accessing the variable must

always reconcile its own private copy of the variable with the master copy in

memory. Say what? Don't worry about it. For the exam, all you need to know about

volatile is that, as with transient, it can be applied only to instance variables.

Make no mistake, the idea of multiple threads accessing an instance variable is scary

stuff, and very important for any Java programmer to understand. But as you'll see in

Chapter 9, you'll probably use synchronization, rather than the volatile modifier,

to make your data thread-safe.

**Immutable**

**Collections**

The core collection interfaces.



* How many elements – **size isEmpty**
* Check whether a given object is in the collection (**contains**)
* Add and remove an element from the collection (**add, remove**)
* Provide an iterator over the collection (**iterator**).

## Traversing Collections

There are three ways to traverse collections:

(1) using aggregate operations

(2) with the for-each construct

(3) by using Iterators.

Note that Iterator.remove is the *only* safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

**for**(String t : names)  
 System.***out***.println(**"ForEach = ["** + t + **"]"**);  
  
Iterator<String> it = names.iterator();  
**while**(it.hasNext()) {  
 System.***out***.println(**"Iterator = ["** + it.next() + **"]"**);  
 it.remove();  
}  
System.***out***.println(**"has elements = ["** + it.hasNext() + **"]"**);

**Set Interface**

HashSet - which stores its elements in a hash table, is the best-performing implementation, however it makes no guarantees concerning the order of iteration.  
TreeSet -  which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet  
LinkedHashSet. which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided by HashSet at a cost that is only slightly higher.

**Iterator**

Iterator takes the place of [Enumeration](https://docs.oracle.com/javase/7/docs/api/java/util/Enumeration.html) in the Java Collections Framework. Iterators differ from enumerations:

Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

ListIterator - allows you to traverse the list in either direction.  
The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces. The hasPrevious and the previous operations are exact analogues of hasNext and next

ListIterator<String> lit = names.listIterator(names.size());  
**while**(lit.hasPrevious()) {  
 System.***out***.println(**"Name = ["** + lit.previous() + **"]"**);  
}

Note the argument to listIterator in the preceding idiom. The List interface has two forms of the listIterator method. The form with no arguments returns a ListIterator positioned at the beginning of the list; the form with an int argument returns a ListIterator positioned at the specified index.

**HashMap vs Hashtable**

There are several differences between HashMap and Hashtable in Java:

1. [Hashtable](http://java.sun.com/javase/7/docs/api/java/util/Hashtable.html) is synchronized, whereas [HashMap](http://java.sun.com/javase/7/docs/api/java/util/HashMap.html) is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
2. Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
3. One of HashMap's subclasses is [LinkedHashMap](http://java.sun.com/javase/7/docs/api/java/util/LinkedHashMap.html), so in the event that you'd want predictable iteration order (which is insertion order by default), you could easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.

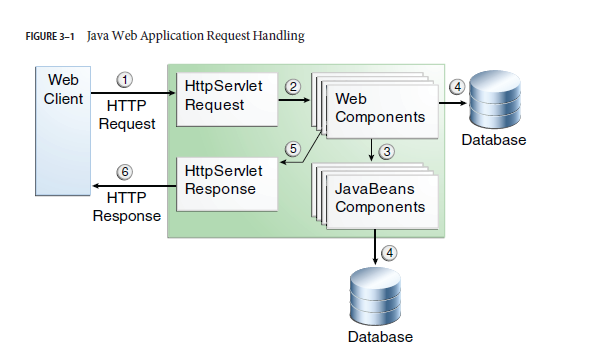
Since synchronization is not an issue for you, I'd recommend HashMap.

As of the Java 2 platform v1.2, this class was retrofitted to implement the [Map](https://docs.oracle.com/javase/7/docs/api/java/util/Map.html) interface, making it a member of the [Java Collections Framework](https://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html). Unlike the new collection implementations, Hashtable is synchronized. If a thread-safe implementation is not needed, it is recommended to use [HashMap](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) in place of Hashtable. If a thread-safe highly-concurrent implementation is desired, then it is recommended to use [ConcurrentHashMap](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html) in place of Hashtable.

**Difference between Comparable and Comparator Interface.**

* a.compareTo(b):  
  **Comparable interface :** Compares values and returns an int which tells if the values compare less than, equal, or greater than.  
  If your class objects have a **natural order**, implement the Comparable<T> interface and define this method. All Java classes that have a natural ordering implement Comparable<T> - Example: String, [wrapper classes](http://www.javatpoint.com/wrapper-class-in-java), BigInteger
* compare(a, b):  
  **Comparator interface :** Compares values of two objects. This is implemented as part of the Comparator<T> interface, and the **typical use is to define one or more small utility classes that implement this, to pass to methods such as sort() or for use by sorting data structures such as TreeMap and TreeSet**. You might want to create a Comparator object for the following:
  + **Multiple comparisons**. To provide several different ways to sort something. For example, you might want to sort a Person class by name, ID, age, height, ... You would define a Comparator for each of these to pass to the sort() method.
  + **System class** To provide comparison methods for classes that you have no control over. For example, you could define a Comparator for Strings that compared them by length.
  + **Strategy pattern** To implement a Strategy pattern, which is a situation where you want to represent an algorithm as an object that you can pass as a parameter, save in a data structure, etc.

**Web Tier**

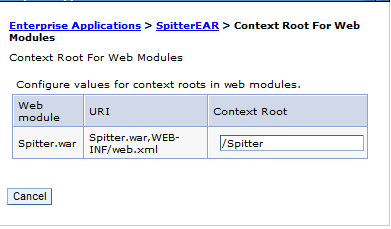


## WEB-INF

Can contain the following files and directories:

* classes: A directory that contains server-side classes: servlets, enterprise bean class files, utility classes, and JavaBeans components.
* lib: A directory that contains JAR files that contain enterprise beans, and JAR archives of libraries called by server-side classes
* Deployment descriptors, such as web.xml (the web application deployment descriptor) and ejb-jar.xml (an EJB deployment descriptor)

**Context Root**



**Web.xml**

The /WEB-INF/web.xml file is the Web Application Deployment Descriptor of your application. This file is an XML document that defines everything about your application that a server needs to know (except the context path, which is assigned by the Application Deployer and Administrator when the application is deployed): servlets and other components like filters or listeners, initialization parameters, container-managed security constraints, resources, welcome pages, etc.

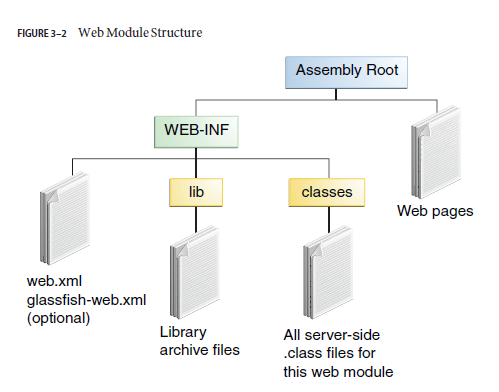
The default web page (the page you get automatically sent to when you don't enter a web page in the URL):

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

 <!-- Set timeout to 120 minutes -->   
 <[session-config](http://wiki.metawerx.net/wiki/Web.xml.SessionConfig)>   
     <[session-timeout](http://wiki.metawerx.net/wiki/Web.xml.SessionTimeout)>120</[session-timeout](http://wiki.metawerx.net/wiki/Web.xml.SessionTimeout)>   
 </[session-config](http://wiki.metawerx.net/wiki/Web.xml.SessionConfig)>



## Applet vs Servlet

Applet runs on the client, servlet runs on the server. It's as simple as that.

More specifically, that applet is downloaded to the client, and executes in a JRE inside the browser, and can display whatever it wants to display within the applet frame. The servlet instead runs on the server and (generally) produces a HTML page which is displayed in your browser.

## What Is a Servlet?

A servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines lifecycle methods. When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. The HttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.

Web application developers typically write servlets that extend javax.servlet.http.HttpServlet, an abstract class that implements the Servlet interface and is specially designed to handle HTTP requests.

## Servlet Lifecycle

The lifecycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps.

1. If an instance of the servlet does not exist, the web container
   1. Loads the servlet class.
   2. Creates an instance of the servlet class.
   3. Initializes the servlet instance by calling the init method.
2. Invokes the service method, passing request and response objects.
3. If it needs to remove the servlet, the container finalizes the servlet by calling the servlet’s destroy method.

**Table 15-1 Servlet Lifecycle Events**

|  |  |  |
| --- | --- | --- |
| **Object** | **Event** | **Listener Interface and Event Class** |
| Web context | Initialization and destruction | javax.servlet.ServletContextListener and ServletContextEvent |
| Web context | Attribute added, removed, or replaced | javax.servlet.ServletContextAttributeListener andServletContextAttributeEvent |
| Session | Creation, invalidation, activation, passivation, and timeout | javax.servlet.http.HttpSessionListener, javax.servlet.http.HttpSessionActivationListener, and HttpSessionEvent |
| Session | Attribute added, removed, or replaced | javax.servlet.http.HttpSessionAttributeListener and HttpSessionBindingEvent |
| Request | A servlet request has started being processed by web components | javax.servlet.ServletRequestListener and ServletRequestEvent |
| Request | Attribute added, removed, or replaced | javax.servlet.ServletRequestAttributeListener and ServletRequestAttributeEvent |

**Creating and Initializing a Servlet**

Use the @WebServlet annotation to define a servlet component in a web application.

@WebServlet("/greeting")

**public** **class** FirstHttpServlet **extends** HttpServlet {

**Writing Service Methods**

The general pattern for a service method is to extract information from the request, access external resources, and then populate the response, based on that information. For HTTP servlets, the correct procedure for populating the response is to do the following:

1. Retrieve an output stream from the response.

2. Fill in the response headers.

3. Write any body content to the output stream.

GenericServlet 🡪 service method  
HttpServlet 🡪 do*Method* (where *Method* can take the value Get, Delete, Options, Post, Put, or

Trace)

**public** **void** doGet(HttpServletRequest request,

HttpServletResponse response)

**throws** ServletException, IOException{

response.setContentType("text/html");

PrintWriter out = response.getWriter();

out.println(message);

**Filters**

A *filter* is an object that can transform the header and content (or both) of a request or response.

Filters differ from web components in that filters usually do not themselves create a response.

The main tasks that a filter can perform are as follows:

* Query the request and act accordingly.
* Block the request-and-response pair from passing any further.
* Modify the request headers and data. You do this by providing a customized version of the request.
* Modify the response headers and data. You do this by providing a customized version of the response.
* Interact with external resources.

Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, XML transformations, and so on.

You define a filter by implementing the Filter interface.  
Classes annotated with the @WebFilter annotation must implement the javax.servlet.Filter interface.

@WebFilter(urlPatterns="/\*", initParams = {

@WebInitParam(name="mood", value="awake")})

**public** **class** TimeofDayFilter **implements** Filter{

The most important method in the Filter interface is doFilter, which is passed request, response, and filter chain objects.

@Override

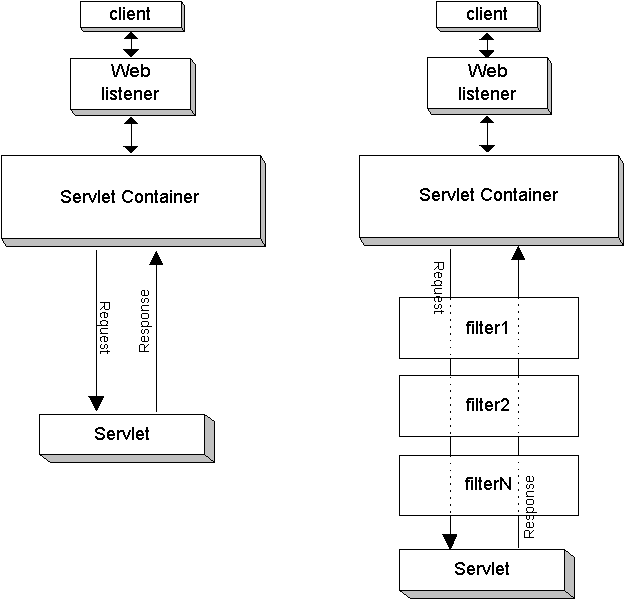
**public** **void** doFilter(ServletRequest request, ServletResponse response,

FilterChain chain) **throws** IOException, ServletException {

**Order of Filters**

If we define webapp specific servlet filters in WAR's own web.xml, then the order of execution of the filters will be the same as the order in which they are defined in the web.xml.

But, if we define those filters using @WebFilter annotation, the order of filter cannot be determined.



**Error Handling:**

<servlet>

<servlet-name>ErrorHandler</servlet-name>

<servlet-class>com.habuna.spitter.ErrorHandler</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>ErrorHandler</servlet-name>

<url-pattern>/ErrorHandler</url-pattern>

</servlet-mapping>

<error-page>

<error-code>404</error-code>

<location>/ErrorHandler</location>

</error-page>

<error-page>

<error-code>403</error-code>

<location>/ErrorHandler</location>

</error-page>

<error-page>

<exception-type>java.lang.Throwable</exception-type >

<location>/ErrorHandler</location>

</error-page>

</web-app>

## Event Listeners

The servlet specification includes the capability to track key events in your Web applications through *event listeners*. This functionality allows more efficient resource management and automated processing based on event status. The following sections describe servlet event listeners:

**Cookies**

Cookies are text files stored on the client computer and they are kept for various information tracking purpose. Java Servlets transparently supports HTTP cookies.

There are three steps involved in identifying returning users:

* Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
* Browser stores this information on local machine for future use.
* When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

Cookie cookie = **new** Cookie("test", "value" );

cookie.setDomain("vineeth.com");

cookie.setMaxAge(60\*60\*100);

**Session Management**

HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.

Still there are following three ways to maintain session between web client and web server:

* Cookies
* Hidden Forms
* URL Re-Writing

The session management techniques we have looked at so far all have a common security issue: they transmit data in plain text. A powerful session-tracking solution is needed that's more secure and flexible. This is where the Java HttpSession API comes in. The HttpSession API provides a simple mechanism for storing information about individual users on the application server.

A session object is created on the application server, usually in a Java servlet or a JavaServer Page. The object is stored on the application server and a unique identifier called a session ID is assigned to it. The session object and session ID are handled by a session manager on the application server. Each session ID assigned by the application server has zero or more key/value pairs tied to it. The values are objects that you place in the session. Assign each of those objects a name, and each name must have an object with it because a null is not allowed.

For this session-tracking technique to work, the session ID must be sent to the client's computer. A cookie is used to store the session ID on the Web site visitor's computer. This is automatically handled by the application server. Simply create the session object and begin using it. The application server will, by default, create the session ID and store it in a cookie

<session-config>  
 <session-timeout>15</session-timeout>  
 </session-config>

**File Upload**

* The form **method** attribute should be set to **POST** method and GET method can not be used.
* The form **enctype** attribute should be set to **multipart/form-data**.
* The form **action** attribute should be set to a servlet file which would handle file uploading at backend server. Following example is using **UploadServlet** servlet to upload file.

The moral of the story is, if you have binary (non-alphanumeric) data (or a significantly sized payload) to transmit, use multipart/form-data. Otherwise, use application/x-www-form-urlencoded.

The following code fragment demonstrates typical usage.

**boolean** isMultipart = ServletFileUpload.*isMultipartContent*(request);

**if**(isMultipart) {

DiskFileItemFactory factory = **new** DiskFileItemFactory();

ServletFileUpload upload = **new** ServletFileUpload(factory);

List<FileItem> fileItems = upload.parseRequest(request);

**for**(FileItem fileitem : fileItems) {

System.*out*.println("Writing File" + fileitem.getName());

**if**(!fileitem.isFormField()) {

File uploadedFile = **new** File("C:\\Temporary\\testwsp\\uploadedfiles\\test.txt");

fileitem.write(uploadedFile);

**@WebListener**

Use the @WebListener annotation to define a listener to get events for various operations on the

particular web application context. Classes annotated with @WebListener must implement one

of the following interfaces:

javax.servlet.ServletContextListener

javax.servlet.ServletContextAttributeListener

javax.servlet.ServletRequestListener

javax.servlet.ServletRequestAttributeListener

javax.servlet..http.HttpSessionListener

javax.servlet..http.HttpSessionAttributeListener

**Table 15-2 Scope Objects**

|  |  |  |
| --- | --- | --- |
| **Scope Object** | **Class** | **Accessible from** |
| Web context | javax.servlet.ServletContext | Web components within a web context. See [Accessing the Web Context](http://docs.oracle.com/javaee/6/tutorial/doc/bnagl.html). |
| Session | javax.servlet.http.HttpSession | Web components handling a request that belongs to the session. See [Maintaining Client State](http://docs.oracle.com/javaee/6/tutorial/doc/bnagm.html). |
| Request | Subtype of javax.servlet.ServletRequest | Web components handling the request. |
| Page | javax.servlet.jsp.JspContext | The JSP page that creates the object. |

## ServletContext

When the servlet container (like [Apache Tomcat](http://tomcat.apache.org/)) starts up, it will deploy and load all its web applications. When a web application is loaded, the servlet container creates the [ServletContext](http://docs.oracle.com/javaee/7/api/javax/servlet/ServletContext.html)once and keeps it in the server's memory. The web app's web.xml file is parsed, and each <servlet>, <filter> and <listener> found (or each class annotated with @WebServlet, @WebFilter and @WebListener respectively) is instantiated once and kept in the server's memory as well. For each instantiated filter, its init() method is invoked immediately.

When the servlet container shuts down, it unloads all web applications, invokes the destroy()method of all its initialized servlets and filters, and all ServletContext, Servlet, Filter and Listener instances are trashed.

When a Servlet has a <servlet><load-on-startup> or @WebServlet(loadOnStartup) value greater than 0, its init() method is also immediately invoked during startup. Those servlets are initialized in the same order specified by that value (1 -> 1st, 2 -> 2nd, etc). If the same value is specified for more than one servlet, then each of those servlets is loaded in the order they appear in the web.xml, or @WebServlet classloading. In the event the "load-on-startup" value is absent, the init() method will be invoked whenever the HTTP request hits that servlet for the very first time.

## HttpServletRequest and HttpServletResponse

The servlet container is attached to a web server that listens for HTTP requests on a certain port number (port 8080 is usually used during development and port 80 in production). When a client (user with a web browser) sends an HTTP request, the servlet container creates new [HttpServletRequest](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html)and [HttpServletResponse](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletResponse.html) objects and passes them through any defined Filter chain and, eventually, the Servlet instance.

In the case of [filters](http://stackoverflow.com/tags/servlet-filters/info), the doFilter() method is invoked. When its code calls chain.doFilter(request, response), the request and response continue on to the next filter, or hit the servlet if there are no remaining filters.

In the case of [servlets](http://stackoverflow.com/tags/servlets/info), the service() method is invoked. By default, this method determines which one of the doXxx() methods to invoke based off of request.getMethod(). If the determined method is absent from the servlet, then an HTTP 405 error is returned in the response.

The request object provides access to all of the information about the HTTP request, such as its headers and body. The response object provides the ability to control and send the HTTP response the way you want by, for instance, allowing you to set the headers and the body (usually with generated HTML content from a JSP file). When the HTTP response is committed and finished, both the request and response objects are recycled and made for reuse.

## HttpSession

When a client visits the webapp for the first time and/or the [HttpSession](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpSession.html) is obtained for the first time via request.getSession(), the servlet container creates a new HttpSession object, generates a long and unique ID (which you can get by session.getId()), and store it in the server's memory. The servlet container also sets a [Cookie](http://docs.oracle.com/javaee/7/api/javax/servlet/http/Cookie.html) in the Set-Cookie header of the HTTP response with JSESSIONID as its name and the unique session ID as its value.

As per the [HTTP cookie specification](http://www.faqs.org/rfcs/rfc2965.html) (a contract a decent web browser and web server have to adhere to), the client (the web browser) is required to send this cookie back in subsequent requests in the Cookie header for as long as the cookie is valid (i.e. the unique ID must refer to an unexpired session and the domain and path are correct). Using your browser's built-in HTTP traffic monitor, you can verify that the cookie is valid (press F12 in Chrome / Firefox 23+ / IE9+, and check the Net/Network tab). The servlet container will check the Cookie header of every incoming HTTP request for the presence of the cookie with the name JSESSIONID and use its value (the session ID) to get the associated HttpSession from server's memory.

The HttpSession stays alive until it has not been used for more than the timeout value specified in <session-timeout>, a setting in web.xml. The timeout value defaults to 30 minutes. So, when the client doesn't visit the web app for longer than the time specified, the servlet container trashes the session. Every subsequent request, even with the cookie specified, will not have access to the same session anymore; the servlet container will create a new session.

On the client side, the session cookie stays alive for as long as the browser instance is running. So, if the client closes the browser instance (all tabs/windows), then the session is trashed on the client's side. In a new browser instance, the cookie associated with the session wouldn't exist, so it would no longer be sent. This causes an entirely new HTTPSession to be created, with an entirely new session cookie begin used.

## In a nutshell

* The ServletContext lives for as long as the web app lives. It is shared among all requests in all sessions.
* The HttpSession lives for as long as the client is interacting with the web app with the same browser instance, and the session hasn't timed out at the server side. It is shared among all requests in the same session.
* The HttpServletRequest and HttpServletResponse live from the time the servlet receives an HTTP request from the client, until the complete response (the web page) has arrived. It is not shared elsewhere.
* All Servlet, Filter and Listener instances live as long as the web app lives. They are shared among all requests in all sessions.
* Any attribute that is defined in ServletContext, HttpServletRequest and HttpSession will live as long as the object in question lives. The object itself represents the "scope" in bean management frameworks such as JSF, CDI, Spring, etc. Those frameworks store their scoped beans as an attribute of its closest matching scope.

## Thread Safety

That said, your major concern is possibly thread safety. You should now know that servlets and filters are shared among all requests. That's the nice thing of Java, it's multithreaded and different threads (read: HTTP requests) can make use of the same instance. It would otherwise be too expensive to recreate, init() and destroy() them for every single request.

You should also realize that you should **never** assign any request or session scoped data as an instance variable of a servlet or filter. It will be shared among all other requests in other sessions. That's **not** thread-safe! The below example illustrates this:

public class ExampleServlet extends HttpServlet {

private Object thisIsNOTThreadSafe;

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

Object thisIsThreadSafe;

thisIsNOTThreadSafe = request.getParameter("foo"); // BAD!! Shared among all requests!

thisIsThreadSafe = request.getParameter("foo"); // OK, this is thread safe.

}

}

**Servlets and JavaServer Pages (JSPs).** The servlets framework is designed to

handle all the infrastructure of deploying a web application and dispatching requests

from remote HTTP clients. A request arriving at the server is dispatched,

perhaps through a chain of filters, to the appropriate servlet or JSP. Each servlet

represents a component of application logic, and in high-volume web sites, multiple

clients may require the services of the same servlet at once. The servlets

specification requires that a servlet be prepared to be called simultaneously from

multiple threads. In other words, servlets need to be thread-safe.

Even if you could guarantee that a servlet was only called from one thread

at a time, you would still have to pay attention to thread safety when building

a web application. Servlets often access state information shared with other

servlets, such as application-scoped objects (those stored in the ServletContext)

or session-scoped objects (those stored in the per-client HttpSession). When a

servlet accesses objects shared across servlets or requests, it must coordinate access

to these objects properly, since multiple requests could be accessing them

simultaneously from separate threads. Servlets and JSPs, as well as servlet filters

and objects stored in scoped containers like ServletContext and HttpSession,

simply have to be thread-safe

**IBM HTTP Server**

IBM® HTTP Server is a full-featured web server that is included with other products such as IBM WebSphere® Application Server at no charge. You can use this web server for projects that do not warrant the expense of a priced and supported HTTP server. The IBM HTTP Server is based on the Apache HTTP Server and provides a rich set of Apache features in addition to IBM enhancement

Oracle **iPlanet Web Server** (OiWS) is a **web server** designed for medium and large business applications. Oracle **iPlanet Web Server** builds on the earlier Sun Java System **Web Server**, Sun ONE **Web Server**, **iPlanet Web Server**, and Netscape Enterprise **Server** products.

*For generating PDF from XML– Apache FOP (Formatting Objects Processor)*

**HTTP Requests**

A request line has three parts, separated by spaces: a *method* name, the local path of the requested resource, and the version of HTTP being used. A typical request line is:

GET /path/to/file/index.html HTTP/1.0

**Initial Response Line (Status Line)**

HTTP/1.0 200 OK

or

HTTP/1.0 404 Not Found

* The HTTP version is in the same format as in the request line, "**HTTP/x.x**".
* The status code is meant to be computer-readable; the reason phrase is meant to be human-readable, and may vary.
* The status code is a three-digit integer, and the first digit identifies the general category of response:
  + **1xx** indicates an informational message only
  + **2xx** indicates success of some kind
  + **3xx** redirects the client to another URL
  + **4xx** indicates an error on the client's part
  + **5xx** indicates an error on the server's part

The most common status codes are:

* **200 OK**
* The request succeeded, and the resulting resource (e.g. file or script output) is returned in the message body.
* **404 Not Found**
* The requested resource doesn't exist.
* **301 Moved Permanently   
  302 Moved Temporarily   
  303 See Other** *(HTTP 1.1 only)*
* The resource has moved to another URL (given by the **Location:** response header), and should be automatically retrieved by the client. This is often used by a CGI script to redirect the browser to an existing file.
* **500 Server Error**
* An unexpected server error. The most common cause is a server-side script that has bad syntax, fails, or otherwise can't run correctly.

## Sample HTTP Exchange

To retrieve the file at the URL

http://www.somehost.com/path/file.html

first open a socket to the host **www.somehost.com**, port 80 (use the default port of 80 because none is specified in the URL). Then, send something like the following through the socket:

GET /path/file.html HTTP/1.0

From: someuser@jmarshall.com

User-Agent: HTTPTool/1.0

[blank line here]

The server should respond with something like the following, sent back through the same socket:

HTTP/1.0 200 OK

Date: Fri, 31 Dec 1999 23:59:59 GMT

Content-Type: text/html

Content-Length: 1354

<html>

<body>

<h1>Happy New Millennium!</h1>

(more file contents)

.

.

.

</body>

</html>

After sending the response, the server closes the socket.

**Spring**

**How does Spring simplify Java Development?**

* Lightweight and minimally invasive development with Plain Old Java Objects (POJOs).
* Loose coupling through dependency injection and Interface Orientation.
* Declarative programming through aspects and common conventions.
* Boilerplate reduction through aspects and templates.

**<context:component-scan>**

By default, <context:component-scan> looks for classes that are annotated with one of a handful of special stereotype annotations:

@Component—A general-purpose stereotype annotation indicating that the class is a Spring component @Controller—Indicates that the class defines a Spring MVC controller   
@Repository—Indicates that the class defines a data repository  
 @Service—Indicates that the class defines a service  
 Any custom annotation that is itself annotated with @Component

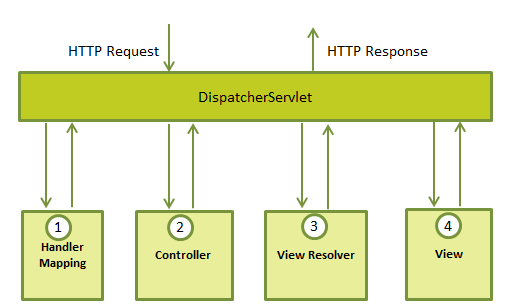
**AspectOriented Programming**

Aspects help to modularize cross-cutting concerns.  
In software development, functions that span multiple points of an application are called *cross-cutting concerns*. Typically, these cross-cutting concerns are conceptually separate from (but often embedded directly within) the application’s business logic. Separating these cross-cutting concerns from the business logic is where aspect- oriented programming (AOP) goes to work.

**Spring MVC**

<mvc:annotation-driven/> - To flip on all of the annotation-driven features

DispatcherServlet



**JAVA Web services**

## There are two approaches to Web service development: bottom-up and top-down.

1. Bottom-up Approach: Starting from Java
2. Top-down Approach: Starting from WSDL

## What is the difference between:

1. [JAX-WS](https://jcp.org/en/jsr/detail?id=224)
2. [Axis2](http://axis.apache.org/axis2/java/core/)
3. [CXF](http://cxf.apache.org/)
4. All three can be used to create webservices in Java.
5. The JAX-WS implementation built into the JDK really is just the basic soap stuff. If you need any of the more complex WS-\* things like WS-Security, WS-RM, WS-Policy, etc..., you need to use one of the alternatives like CXF or Metro or Axis2.
6. CXF also has other things besides just JAX-WS. It has a compliant JAX-RS implementation as well and supports exposing services as both REST and SOAP very well.
7. CXF has top notch Spring support

### Advantages and Disadvantages of SOAP and REST styles.

1. REST is an architectural style, but SOAP is a protocol.
2. RESTs sweet spot is when you are exposing a public API over the internet to handle CRUD operations on data.
3. REST is focused on accessing named resources through a single consistent interface.
4. Since REST uses standard HTTP it is much simpler in just about ever way. Creating clients, developing APIs, the documentation is much easier to understand and there aren’t very many things that REST doesn’t do easier/better than SOAP.
5. REST permits many different data formats where as SOAP only permits XML.
6. JSON usually is a better fit for data and parses much faster.
7. REST has better performance and scalability. REST reads can be cached, SOAP based reads cannot be cached.
8. However, using RESTful service APIs is akin to doing meta-programming like Ruby and Python which delays problems to run-time as there is no defined schema agreed and technically enforced upon by the two sides. As such I don't recommend RESTful services everywhere, but I would recommend it if I had control of the two sides which normally happens when you do build a web application that uses static HTML/CSS/JS and talks with a RESTful server for the data.

## Difference between JAX-WS vs JAX-RS

1. We can create RESTful webservices from JAX-WS also. But most of people says that we should use JAX-RS instead of JAX-WS.
2. JAX-WS is generally geared towards server to server interactions with well defined contracts (WSDLs) and usually when the service and client side are from separate groups. It is very resource intensive so it isn't feasible for client-to-server interactions where the network or client device capability is less than optimal.
3. JAX-RS is geared towards client to server interactions, although server-to-server is okay. The only contractual obligation between a client and server is the message and the request headers. As it has little service obligations, it can be tuned to whatever the client needs are.

**Spring Security**

Add Filter in web.xml

**Cache Control headers**

– informs the browser/proxy to avoid storing the HTML response.

* Cache control
* Pragma
* Expires

**REST API Security**

1. Citi connect APIs are exposed as REST APIs, protected by mutual certificate authentication and OAuth 2.0 authorization protocol. Citi connect APIs currently support Partner credentials grant type of available OAuth 2.0 grant types. Below are the details for Authentication API.

# Securing an API by using OAuth 2.0

OAuth provides a method for users to grant third-party access to their resources without sharing their passwords. It also provides a way to grant limited access (in scope, duration, etc.).

[**https://hueniverse.com/oauth/guide/workflow/**](https://hueniverse.com/oauth/guide/workflow/)

**Threads**

You can create threads in two ways:

1. Extends **Thread – not recommended**
2. Implements **Runnable** and pass an instance of your news class to the Thread constructor

**public class** MyFirstRunnable **implements** Runnable{  
  
  
 **public void** run() {  
 System.***out***.println(**"Im thread :"** + Thread.*currentThread*().getName());  
 }  
}

MyFirstRunnable runnable = **new** MyFirstRunnable();

Thread runnable = new Thread(new MyFirstRunnable());

**Thread.join()**

To quote from the [Thread.join() method javadocs](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.html#join%28%29):

Waits for this thread to die.

There is a thread that is running your example code which is probably the [main thread](http://www.go4expert.com/articles/main-thread-java-t4178/).

1. The main thread creates and starts the t1 and t2 threads. The two threads start running in parallel.
2. The main thread calls t1.join() to wait for the t1 thread to finish.
3. The t1 thread completes and the t1.join() method returns in the main thread.
4. The main thread calls t2.join() to wait for the t2 thread to finish.
5. The t2 thread completes (or completed before the t1 thread did) and the t2.join() method returns in the main thread.

It is important to understand that the t1 and t2 threads have been running **in parallel** but the main thread that started them needs to wait for them to finish before it can continue. That's a common pattern.

t1.join() means cause t2 to stop until t1 terminates?

No. The main thread that is calling t1.join() will stop running and wait for the t1 thread to finish. The t2 thread is running in parallel and is not affected by t1 or the t1.join() call at all.

In terms of the try/catch, the join() throws InterruptedException meaning that the main thread that is calling join() may itself be interrupted by another thread.

**Inside JVM**

* When you write and run a Java program, you are tapping the power of these four technologies. You express the program in source files written in the Java programming language.
* Compile the source to Java class files, and run the class files on a Java Virtual Machine.
* When you write your program, you access system resources (such as I/O, for example) by calling methods in the classes that implement the Java Application Programming Interface, or Java API.
* As your program runs, it fulfills your program’s Java API calls by invoking methods in class files that implement the Java API. You can see the relationship between these four parts in Figure 1-1.



* A Java Virtual Machine’s main job is to load class files and execute the bytecodes they contain.
* The bytecodes are executed in an *execution engine*, which is one part of the virtual machine that can vary in different implementations.
* The simplest kind of execution engine just interprets the bytecodes one at a time.
* Another kind of execution engine, one that is faster but requires more memory, is a *just-in-time compiler*. In this scheme, the bytecodes of a method are compiled to native machine code the first time the method is invoked. The native machine code for the method is then cached, so it can be re-used the next time that same method is invoked.



Sometimes the Java Virtual Machine is called the *Java interpreter*; however, given the various ways in which bytecodes can be executed, this term can be misleading. While "Java interpreter" is a reasonable name for a Java Virtual Machine that interprets bytecodes, virtual machines also use other techniques (such as just-in-time compiling) to execute bytecodes. Therefore, although all Java interpreters are Java Virtual Machines, not all Java Virtual Machines are Java interpreters.

**Class Loader Architecture**

* There may be more than one class loader inside a Java Virtual Machine.
* The primordial class loader (there is only one of them) is a part of the Java Virtual Machine implementation.
* The Java Virtual Machine considers any class it loads through the primordial class loader to be trusted, regardless of whether or not the class is part of the Java API.
* Classes it loads through class loader objects, however, it views with suspicion--by default, it considers them to be untrusted.



For each class it loads, the Java Virtual Machine keeps track of which class loader--whether primordial or object--loaded the class.

When a loaded class first refers to another class, the virtual machine requests the referenc*ed* class from the same class loader that originally loaded the referenc*ing* class.

For example, if the virtual machine loads class Volcano through a particular class loader, it will attempt to load any classes Volcano refers to through the same class loader. If Volcano refers to a class named Lava, perhaps by invoking a method in class Lava, the virtual machine will request Lava from the class loader object that loaded Volcano. The Lava class returned by the class loader is dynamically linked with class Volcano.

Because the Java Virtual Machine takes this approach to loading classes, classes can by default only see other classes that were loaded by the same class loader. This is how Java’s architecture enables you to create multiple *name-spaces* inside a single Java application. Each class loader in your running Java program maintains its own name-space, which is populated by the names of all the classes it has loaded.

One example of dynamic extension is the web browser, which uses class loader objects to download the class files for an applet across a network. A web browser fires off a Java application that installs a class loader object--usually called an *applet class loader*--that knows how to request class files from an HTTP server.

Applets are an example of dynamic extension, because the Java application doesn’t know when it starts which class files the browser will ask it to download across the network. The class files to download are determined at run-time, as the browser encounters pages that contain Java applets.

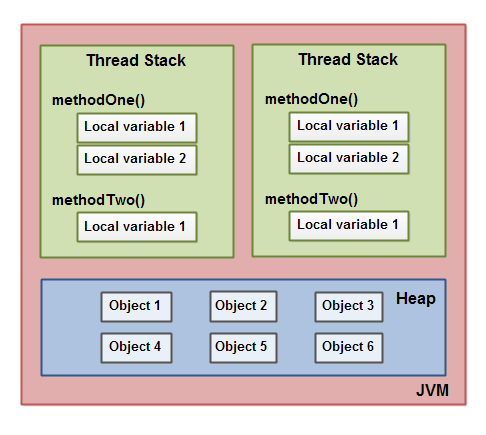
The Java application started by the web browser usually creates a different applet class loader object for each location on the network from which it retrieves class files. As a result, class files from different sources are loaded by different class loader objects. This places them into different name-spaces inside the host Java application. Because the class files for applets from different sources are placed in separate name-spaces, the code of a malicious applet is restricted from interfering directly with class files downloaded from any other source.

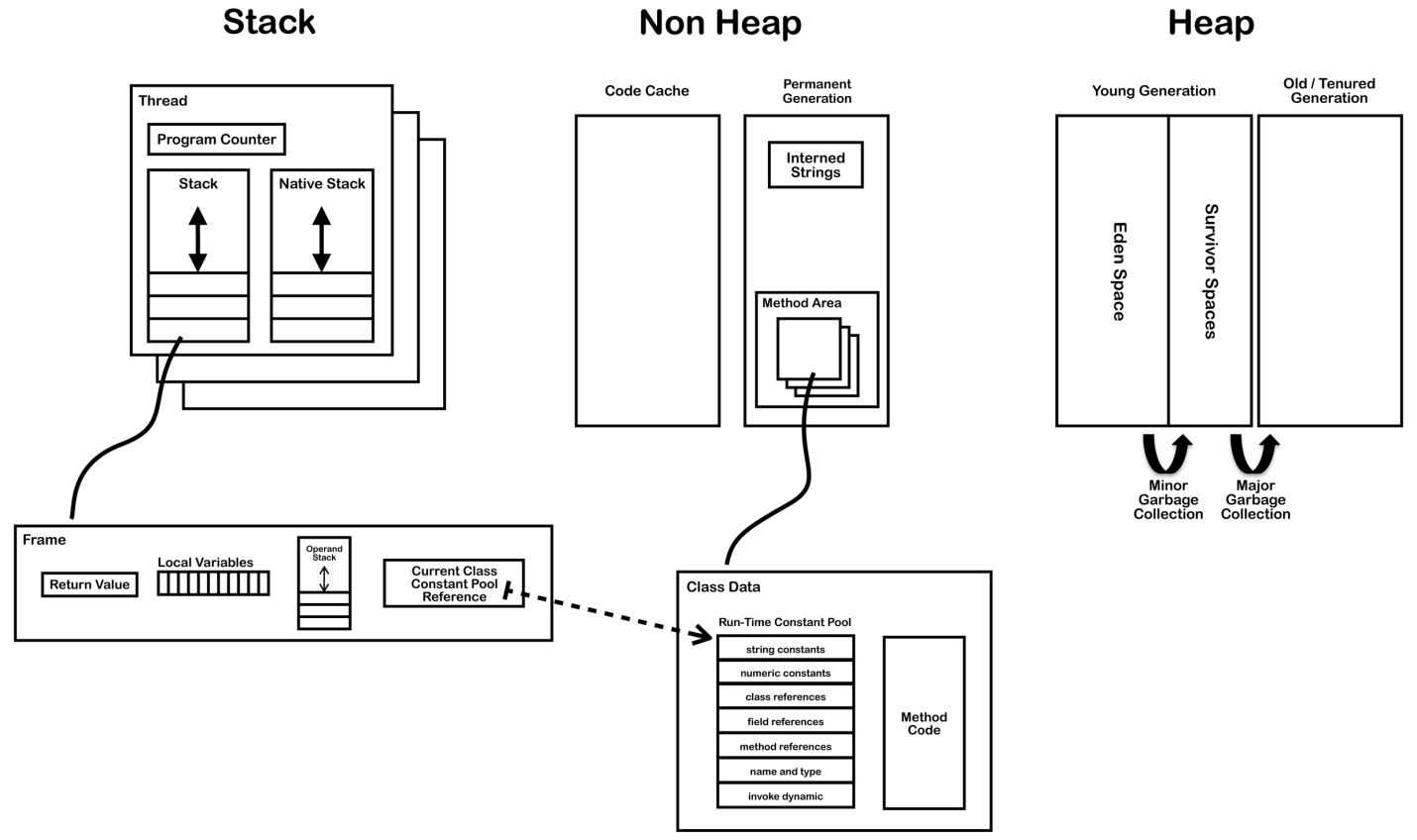
This puts the class files from different sources into different name-spaces, which allows you to restrict or prevent access between code loaded from different sources.

[HotSpot](http://en.wikipedia.org/wiki/HotSpot) is an an implementation of the JVM concept, originally developed by Sun and now owned by Oracle. There are other implementations of the JVM specification, like [JRockit](http://en.wikipedia.org/wiki/JRockit), [IBM J9](http://en.wikipedia.org/wiki/IBM_J9), among many others.

LOT MORE TO LEARN….

**JAVA MEMORY MODEL**





## Thread

A thread is a thread of execution in a program.

* The JVM allows an application to have multiple threads of execution running concurrently.
* In the Hotspot JVM there is a direct mapping between a Java Thread and a native operating system Thread.
* After preparing all of the state for a Java thread such as thread-local storage, allocation buffers, synchronization objects, stacks and the program counter, the native thread is created.
* The native thread is reclaimed once the Java thread terminates.
* The operating system is therefore responsible for scheduling all threads and dispatching them to any available CPU.
* Once the native thread has initialized it invokes the run() method in the Java thread. When the run() method returns, uncaught exceptions are handled, then the native thread confirms if the JVM needs to be terminated as a result of the thread terminating (i.e. is it the last non-deamon thread).
* When the thread terminates all resources for both the native and Java thread are released.

If a thread requires a larger stack than allowed a StackOverflowError is thrown. Main reason Recursion.

[**http://blog.jamesdbloom.com/JVMInternals.html#threads**](http://blog.jamesdbloom.com/JVMInternals.html#threads)