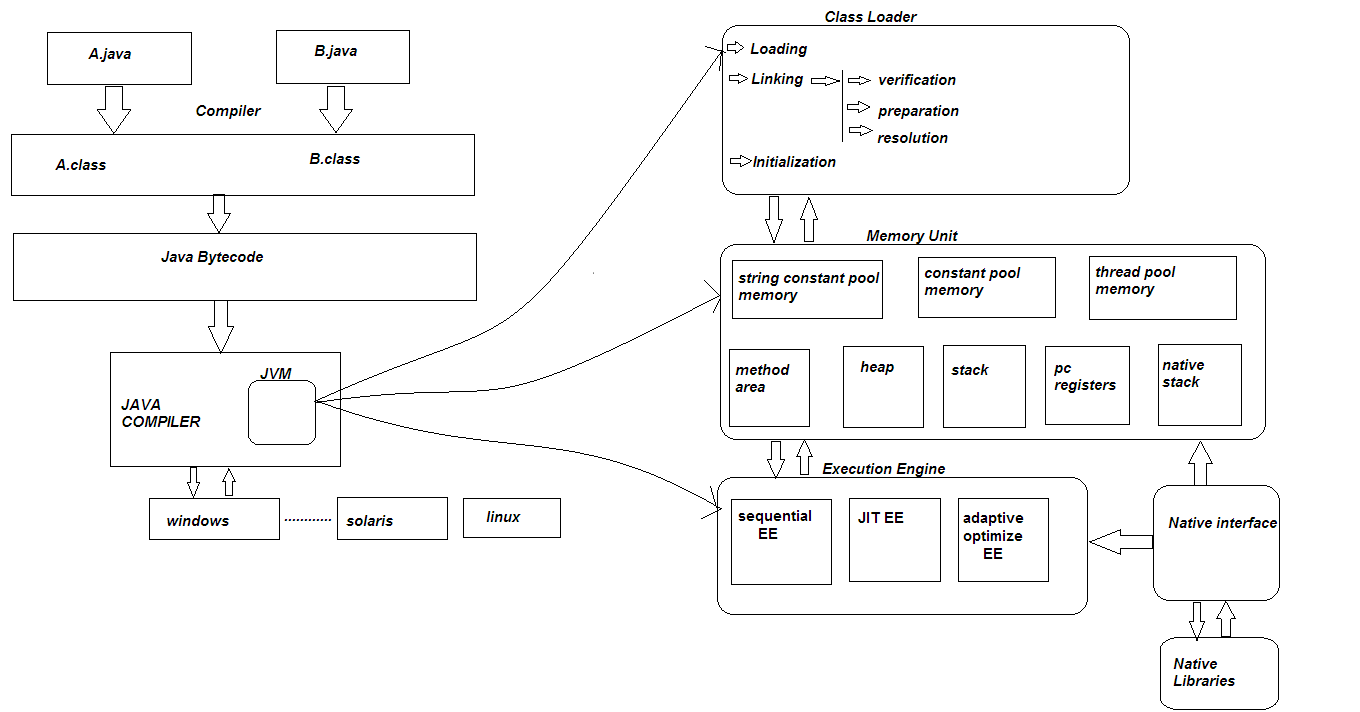


Table 2–1 Sun Java System Application Server Class Loaders

| **Class Loader** | **Description** |
| --- | --- |
| Bootstrap | The Bootstrap class loader loads the basic runtime classes provided by the JVM, plus any classes from JAR files present in the system extensions directory. It is parent to the System class loader. To add JAR files to the system extensions, directory, see [Using the Java Optional Package Mechanism](https://docs.oracle.com/cd/E19501-01/819-3659/beadk/index.html). |
| System | The System class loader loads Application Server launch classes. It is parent to the Shared Chain class loader. It is created based on the system-classpath attribute of the java-config element in the domain.xml file. In the Admin Console, select the Application Server component, the JVM Settings tab, and the Path Settings tab, then edit the System Classpath field. See [Using the System Class Loader](https://docs.oracle.com/cd/E19501-01/819-3659/beadi/index.html) and [*java-config* in *Sun Java System Application Server Platform Edition 9 Administration Reference*](https://docs.oracle.com/docs/cd/E19501-01/819-3661/abhcx/index.html). |
| Shared Chain | The Shared Chain class loader loads most of the core Application Server classes. It is parent to the MBean class loader and the Common class loader. Classes specified by the classpath-prefix and classpath-suffix attributes of thejava-config element in the domain.xml file are added to this class loader. In the Admin Console, select the Application Server component, the JVM Settings tab, and the Path Settings tab, then edit the Classpath Prefix or Classpath Suffix field.  The environment classpath is included if env-classpath-ignored="false" is set in the java-config element.  Use classpath-prefix to place libraries ahead of Application Server implementation classes in the shared chain. Theclasspath-prefix is ideal for placing development and diagnostic patches. Use classpath-suffix to place libraries after implementation classes in the shared chain. |
| MBean | The MBean class loader loads the MBean implementation classes. See [MBean Class Loading](https://docs.oracle.com/cd/E19501-01/819-3659/gbfcp/index.html). |
| Common | The Common class loader loads classes in the *domain-dir*/lib/classes directory, followed by JAR files in the *domain-dir*/lib directory. It is parent to the Connector class loader. No special classpath settings are required. The existence of these directories is optional; if they do not exist, the Common class loader is not created. See [Using the Common Class Loader](https://docs.oracle.com/cd/E19501-01/819-3659/beadj/index.html). |
| Connector | The Connector class loader is a single class loader instance that loads individually deployed connector modules, which are shared across all applications. It is parent to the LifeCycleModule class loader and the Application class loader. |
| LifeCycleModule | The LifeCycleModule class loader is created once per lifecycle module. Each lifecycle-module element’sclasspath attribute is used to construct its own class loader. For more information on lifecycle modules, see [Chapter 13, Developing Lifecycle Listeners](https://docs.oracle.com/cd/E19501-01/819-3659/beamc/index.html). |
| Application | The Application class loader loads the classes in a specific enabled individually deployed module or Java EE application. One instance of this class loader is present in each class loader universe; see [Class Loader Universes](https://docs.oracle.com/cd/E19501-01/819-3659/beadg/index.html). The Application class loader is created with a list of URLs that point to the locations of the classes it needs to load. It is parent to the Web class loader.  The Application class loader loads classes in the following order:   1. Classes specified by the library-directory element in the application.xml deployment descriptor or the–-libraries option during deployment; see [Application-Specific Class Loading](https://docs.oracle.com/cd/E19501-01/819-3659/gatej/index.html) 2. Classes specified by the application's or module's location attribute in the domain.xml file, determined during deployment 3. Classes in the classpaths of the application's sub-modules 4. Classes in the application's or module's stubs directory   The location attribute points to *domain-dir*/applications/j2ee-apps/*app-name* or *domain-dir*/applications/j2ee-modules/*module-name*.  The stubs directory is *domain-dir*/generated/ejb/j2ee-apps/*app-name* or *domain-dir*/generated/ejb/j2ee-modules/*module-name*. |
| Web | The Web class loader loads the servlets and other classes in a specific enabled web module or a Java EE application that contains a web module. This class loader is present in each class loader universe that contains a web module; see [Class Loader Universes](https://docs.oracle.com/cd/E19501-01/819-3659/beadg/index.html). One instance is created for each web module. The Web class loader is created with a list of URLs that point to the locations of the classes it needs to load. The classes it loads are in WEB-INF/classes or WEB-INF/lib/\*.jar. It is parent to the JSP Engine class loader. |
| JSP Engine | The JSP Engine class loader loads compiled JSP classes of enabled JSP files. This class loader is present in each class loader universe that contains a JSP page; see [Class Loader Universes](https://docs.oracle.com/cd/E19501-01/819-3659/beadg/index.html). The JSP Engine class loader is created with a list of URLs that point to the locations of the classes it needs to load. |

<https://docs.oracle.com/cd/E19501-01/819-3659/6n5s6m5b2/index.html>

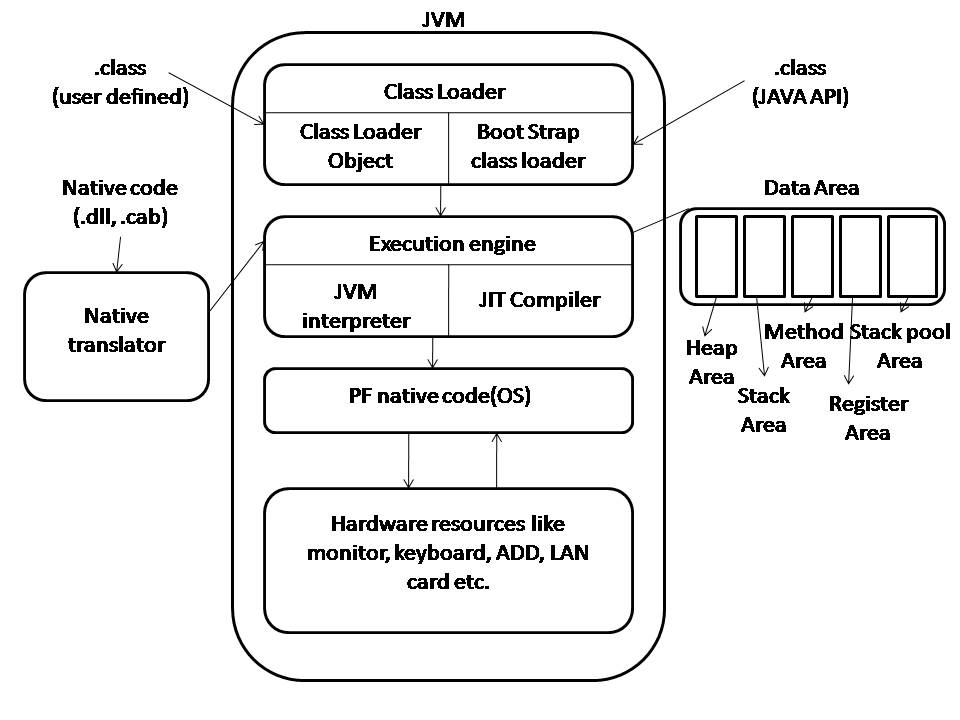


**Constant pool:** When a **Java** file is compiled, all references to variables and methods are stored in the class's**constant pool** as a symbolic reference. A symbolic reference is a logical reference not a reference that actually points to a physical memory location.

The Method [Source](Area(http:/www.artima.com/insidejvm/ed2/jvm.html) Very Important

http://www.artima.com/insidejvm/ed2/jvm.html

Inside a Java virtual machine instance, information about loaded types is stored in a logical area of memory called the method area. When the Java virtual machine loads a type, it uses a class loader to locate the appropriate class file. The class loader reads in the class file--a linear stream of binary data--and passes it to the virtual machine. The virtual machine extracts information about the type from the binary data and stores the information in the method area. Memory for class (static) variables declared in the class is also taken from the method area.



**http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.5**

**The wrapper classes**:

wrapper in java servers two primary purposes.

* To provide mechanism to ‘wrap’ primitive values in an object so that primitives can do activities reserved for the objects like being added to ArrayList, Hashset, HashMap etc. collection.
* To provide an assortment of utility functions for primitives like converting primitive types to and from string objects, converting to various bases like binary, octal or hexadecimal, or comparing various objects.

# [Why String is immutable in Java ?](http://www.programcreek.com/2013/04/why-string-is-immutable-in-java/)

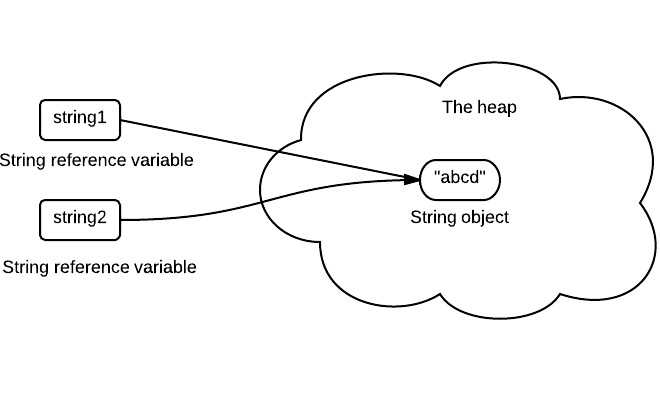
String is an immutable class in Java. An immutable class is simply a class whose instances cannot be modified. All information in an instance is initialized when the instance is created and the information can not be modified. There are many advantages of immutable classes. This article summarizes why [String is designed to be immutable](http://www.programcreek.com/2009/02/diagram-to-show-java-strings-immutability/). A good answer depends on the deep understanding of memory, synchronization, data structures, etc.

**1. Requirement of String Pool**

String pool (String intern pool) is a special storage area in [Method Area](http://www.programcreek.com/2013/04/jvm-run-time-data-areas/). When a string is created and if the string already exists in the pool, the reference of the existing string will be returned, instead of creating a new object and returning its reference.

The following code will create only one string object in the heap.

|  |
| --- |
| String string1 = "abcd";  String string2 = "abcd"; |

Here is how it looks:  
[](http://www.programcreek.com/wp-content/uploads/2013/07/java-string-pool.jpeg)

If a string is not immutable, changing the string with one reference will lead to the wrong value for the other references.

**2. Caching Hashcode**

The hashcode of the string is frequently used in Java. For example, in a HashMap. Being immutable guarantees that hashcode will always the same so that it can be cashed without worrying the changes.That means, there is no need to calculate hashcode every time it is used. This is more efficient.

In String class, it has the following code:

|  |
| --- |
| **private** **int** hash;*//this is used to cache hash code.* |

**3. Facilitating the Use of Other Objects**

To make this concrete, consider the following program:

|  |
| --- |
| HashSet<String> set = **new** HashSet<String>();  set.add(**new** String("a"));  set.add(**new** String("b"));  set.add(**new** String("c"));    **for**(String a: set)  a.value = "a"; |

In this example, if String is mutable, it's value can be changed which would violate the design of set (set contains unduplicated elements). This example is designed for simplicity sake, in the real String class there is no value field.

**4. Security**

String is widely used as a parameter for many java classes, e.g. network connection, opening files, etc. Were String not immutable, a connection or file would be changed and lead to a serious security threat. The method thought it was connecting to one machine, but was not. Mutable strings could cause a security problem in Reflection too, as the parameters are strings.

Here is a code example:

|  |
| --- |
| **boolean** connect(string s){  **if** (!isSecure(s)) {  **throw** **new** SecurityException();  }  *//here will cause problem, if s is changed before this by using other references.*  causeProblem(s);  } |

**5. Immutable objects are naturally thread-safe**

Because immutable objects can not be changed, they can be shared among multiple threads freely. This eliminates the requirements of doing synchronization.

In summary, String is designed to be immutable for the sake of efficiency and security. This is also the reason why immutable classes are preferred in general.

# Why Use Generics?

Code that uses generics has many benefits over non-generic code:

* Stronger type checks at compile time.  
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
* Elimination of casts.  
  The following code snippet without generics requires casting:
* List list = new ArrayList();
* list.add("hello");
* String s = **(String)** list.get(0);

When re-written to use generics, the code does not require casting:

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); // no cast

* Enabling programmers to implement generic algorithms.  
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

**Error:**

public class **Error**

extends [Throwable](https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html)

An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The ThreadDeath error, though a "normal" condition, is also a subclass of Error because most applications should not try to catch it.

A method is not required to declare in its throws clause any subclasses of Error that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur. That is, Error and its subclasses are regarded as unchecked exceptions for the purposes of compile-time checking of exceptions.

# [Why should not we catch Errors in Java?](http://stackoverflow.com/questions/15242207/why-should-not-we-catch-errors-in-java)

The reason that you shouldnt generally attempt to handle these errors is because more often than not there wont be anything at all you can do about them.

They will tend to be JVM level errors, not application level ones - OutOfMemory is a good example here. If the JVM has run out of memory what would your program do? And even if you did catch it there is no guarentee that the handling code would complete/proceed in a consistent manner, given the terminal condition thrown.

OOPs Concepts:

OOPs stands for Object Oriented Programming. The concepts in oops are similar to any other programming languages. Basically, it is program agnostic.

The different OOps concepts are :

* [Polymorphism](http://java-questions.com/oops-interview-questions.html#polymorphism-java)
* [Inheritance](http://java-questions.com/oops-interview-questions.html#inheritance-in-java)
* [Abstraction](http://java-questions.com/oops-interview-questions.html#abstraction)
* [Encapsulation](http://java-questions.com/oops-interview-questions.html#encapsulation)
* [Aggreagation](http://java-questions.com/oops-interview-questions.html#aggregation)
* [Composition](http://java-questions.com/oops-interview-questions.html#composition)
* [Association](http://java-questions.com/oops-interview-questions.html#association)

## Abstraction

In computer science, abstraction is the process by which data and programs are defined with a representation similar in form to its meaning (semantics), while hiding away the implementation details.

In more simple terms, abstraction is to hide information that is not relevant or rather show only relevant information and to simplify it by comparing it to something similar in the real world.

**Encapsulation:**

The encapsulation is achieved by combining the methods and attribute into a class. The class acts like a container encapsulating the properties.

**What is Association?**

Ans) Association is a relationship where all object have their own lifecycle and there is no owner. Let's take an example of Teacher and Student. Multiple students can associate with single teacher and single student can associate with multiple teachers but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.

**Q7) What is Aggregation?**

Ans) Aggregation is a specialize form of Association where all object have their own lifecycle but there is ownership and child object can not belongs to another parent object. Let's take an example of Department and teacher. A single teacher can not belongs to multiple departments, but if we delete the department teacher object will not destroy. We can think about "has-a" relationship.

**Q8) What is Composition ?**

Ans) Composition is again specialize form of Aggregation and we can call this as a "death" relationship. It is a strong type of Aggregation. Child object dose not have their lifecycle and if parent object deletes all child object will also be deleted. Let's take again an example of relationship between House and rooms. House can contain multiple rooms there is no independent life of room and any room can not belongs to two different house if we delete the house room will automatically delete.

# What Is an Object?

Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.

# What Is a Class?

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

# What Is Inheritance?

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes. In this example, Bicycle now becomes the *superclass* of MountainBike, RoadBike, and TandemBike. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*:

**Abstract class and interface which one should use when ?**

Here are some recommendations to help you to decide whether to use an interface or an abstract class to provide polymorphism for your components.

* If you anticipate creating multiple versions of your component, create an abstract class. Abstract classes provide a simple and easy way to version your components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, you must create a whole new interface.
* If the functionality you are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing common functionality to unrelated classes.
* If you are designing small, concise bits of functionality, use interfaces. If you are designing large functional units, use an abstract class.
* If you want to provide common, implemented functionality among all implementations of your component, use an abstract class. Abstract classes allow you to partially implement your class, whereas interfaces contain no implementation for any members.

**String hashCode() implementation:**

**public** **int** hashCode() {

**int** h = hash;//u can assume for new string hash==0

**int** len = count;//char array size or number of character in string

**if** (h == 0 && len > 0) {

**int** off = offset; //string use 0 as offset value

**char** val[] = value;

**for** (**int** i = 0; i < len; i++) {

h = 31\*h + val[off++];

}

hash = h;

}

**return** h;

}

HashMap put method :

**public** V put(K key, V value) {

**if** (key == **null**)

**return** putForNullKey(value);

**int** hash = *hash*(key.hashCode());//call string hashcode() method then call hash() method

**int** i = *indexFor*(hash, table.length);//calculate bucket number using hash XOR table size

**for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) { //linked list implementation

Object k;

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(**this**);

**return** oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

**return** **null**;

}

/\*\*

\* Applies a supplemental hash function to a given hashCode, which

\* defends against poor quality hash functions. This is critical

\* because HashMap uses power-of-two length hash tables, that

\* otherwise encounter collisions for hashCodes that do not differ

\* in lower bits. Note: Null keys always map to hash 0, thus index 0.

\*/

**static** **int** hash(**int** h) {

// This function ensures that hashCodes that differ only by

// constant multiples at each bit position have a bounded

// number of collisions (approximately 8 at default load factor).

h ^= (h >>> 20) ^ (h >>> 12);

**return** h ^ (h >>> 7) ^ (h >>> 4);

}

/\*\*

\* Returns index for hash code h.

\*/

**static** **int** indexFor(**int** h, **int** length) { //bucket number

**return** h & (length-1);

}

**CuncurrentHashMap<K,V> put method:**

V put(K key, **int** hash, V value, **boolean** onlyIfAbsent) {

lock();

**try** {

**int** c = count;

**if** (c++ > threshold) // ensure capacity

rehash();

HashEntry<K,V>[] tab = table;

**int** index = hash & (tab.length - 1);

HashEntry<K,V> first = tab[index];

HashEntry<K,V> e = first;

**while** (e != **null** && (e.hash != hash || !key.equals(e.key)))

e = e.next;

V oldValue;

**if** (e != **null**) {

oldValue = e.value;

**if** (!onlyIfAbsent)

e.value = value;

}

**else** {

oldValue = **null**;

++modCount;

tab[index] = **new** HashEntry<K,V>(key, hash, first, value);

count = c; // write-volatile

}

**return** oldValue;

} **finally** {

unlock();

}

}

**Java 8 Video:**

[**http://www.oracle.com/events/us/en/java8/index.html**](http://www.oracle.com/events/us/en/java8/index.html)

**JVM Internal:**

**http://www.artima.com/insidejvm/ed2/jvmP.html**