*[Difference between Association, Composition and Aggregation in Java, UML and Object Oriented Programming](http://javarevisited.blogspot.in/2014/02/ifference-between-association-vs-composition-vs-aggregation.html" \o "Difference between Association, Composition and Aggregation in Java, UML and Object Oriented Programming)*

In Object-oriented programming, one object is related to other to use functionality and service provided by that object. This relationship between two object is known as *association*in  object oriented general software design, and depicted by an arrow in Unified Modelling language or UML. Both Composition and Aggregation are form of association between two objects, but there is subtle difference between composition and aggregation, which is also reflected by their UML notation. We refer association between two objects as composition, when one class *owns*other class and other class can not meaningfully exist, when it's owner destroyed, for example Human class is composition of several body parts including Hand, Leg and Heart. When human object dies, all it's body part ceased to exist meaningfully, this is one example of Composition. Programmers often confuse between Association, Composition and Aggregation in Object oriented design discussions, this confusion also makes *difference between Association, Composition and Aggregation* one of the popular questions in Java Interviews, only after [difference between abstract class and interface](http://javarevisited.blogspot.com/2013/05/difference-between-abstract-class-vs-interface-java-when-prefer-over-design-oops.html) . Another example of Composition is Car and it's part e.g. engines, wheels etc. Individual parts of car can not function, when car is destroyed.  While in case of Aggregation, including object can exists without being part of main object e.g. a Player which is part of a Team, can exists without team and can become part of other teams as well. Another example ofAggregation is Student in School class, when School closed, Student still exist and then can join another School or so.  In UML notation, composition is denoted by a filled diamond, while aggregation is denoted by an empty diamond, which shows their obvious difference in terms of strength of relationship. Composition is more stronger than Aggregation.  In Short, relationship between two objects is referred as association, and an association is known as composition when one object *owns*other, while an association is known as aggregation when one object uses other object. In this OOPS tutorial, we will see couple of more examples to understand difference between Association, Composition and Aggregation better.  
  
**Difference between abstract class and interface class**

**1) Interfaces have all methods inherently public and abstract. You can not override this behavior by trying to reduce accessibility of methods. You can not even declare the static methods. Only public and abstract.**

**On other side, abstract classes are flexible in declaring the methods. You can define abstract methods with protected accessibility also. Additionally, you can define static methods as well, provided they are not abstract. Non-abstract static methods are allowed.**

**2) Interfaces can’t have fully defined methods. By definition, interfaces are meant to provide only contract.**

**Abstract classes can have non-abstract methods without any limitation. You can use any keyword with non-abstract methods as you will do in any other class.**

**3) Any class which want to use abstract class can extend abstract class using keyword extends, whereas for implementinginterfaces keyword used is implements.**

**A class can extends only one class, but can implement any number of interfaces. This property is often referred as simulation of multiple inheritance in java.**

**4) Interface is absolutely abstract and cannot be instantiated; A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.**

**Next, question may come if we have abstract methods and main class both, we may try to call abstract method from main(). But this attempt will fail, as main() method is always static and abstract methods can never be static, so you can never access any non-static method inside static method.**

**Difference between String and StringBuffer in Java**

**String vs StringBuffer in Java**

**String and StringBuffer are two classes which is most widely used in any Java program. If I say you can not write a program without using String in Java than it would not be exaggeration. String is everywhere, main method accept String argument, logs are String etc. Though many Java programmers familiar with String, not many are careful while performing operations on String, Since String is final in Java; every operation e.g. converting String into Uppercase, creating SubString, converting String to Lowercase all result in a separate new String Object, which can take trigger frequent garbage collection and affect your application performance. here comes StringBuffer in Java which is a mutable version of String, though its not as feature rich as String and you can not use StringBuffer in place of String but StringBuffer should be used whenever you are performing String concatenation instead of String in Java. In this Java tutorial we will see some difference between String and StringBuffer, which makes StringBuffer ideal choice for performing String operations.By the way if you gone through any Java interview then you must be familiar with this question, String vs StringBuffer**

**1) First and most significant difference between String and StringBuffer in Java is that String is immutable in Java while StringBuffer is mutable. What this means is, performing any operation on String will create new String object while modifying StringBuffer object won't create new object.**

**2) If you are using + operator for concatenating multiple String than you should not be worried much because based upon Java implementation call to + operator is replaced with either StringBuffer or StringBuider based upon JVM implements Java 1.5 or lower version.**

**3) StringBuffer.append()method is used to perform String concatenation in Java.**

**4) Creating StringBuffer from String is easy, as StringBuffer accept an String input. Similarly converting Stringbuffer to String is also easy by using toString() method in Java.**

**5)Another significant difference between String and StringBuffer is that StringBuffer and String does not share same type hierarchy, means you can not cast String to StringBuffer in Java. any such attempt will result in ClassCastException in Java.**

**That's all on Difference between String and StringBuffer in Java. Most important difference to remember between String and StringBuffer is mutability.**

An Example of Association, Composition and Aggregation in Java

Here is an example of composition and aggregation, in terms of Java Code. By looking at this code, you can gauge differences between these two. By the way, Composition is also very much preferred in object oriented design over inheritance, even Joshua bloach has stated its importance in classic book, Effective Java.  
  
Composition : Since Engine is part-of Car, relationship between them is Composition. Here is how they are implemented between Java classes.

public class Car {

//final will make sure engine is initialized

private final Engine engine;

public Car(){

engine = new Engine();

}

}

class Engine {

private String type;

}

Aggregation : Since Organization has Person as employees, relationship between them is Aggregation. Here is how they look like in terms of Java classes

public class Organization {

private List employees;

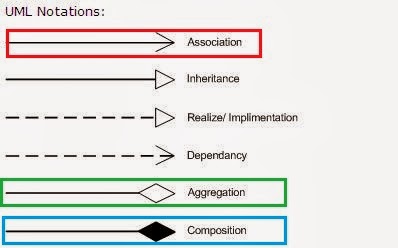
}

public class Person {

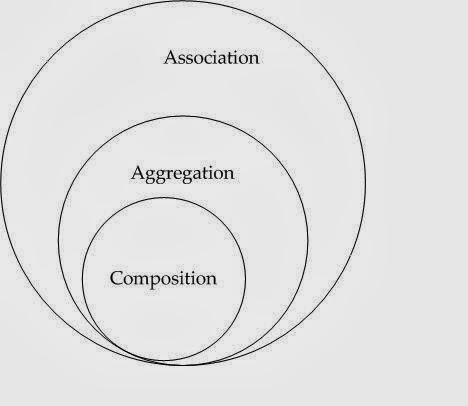
private String name;

}

UML Diagram of Association, Composition and Aggregation   
UML has different notations to denote aggregation, composition and association.  Association is denoted by simple arrow, while aggregation is denoted by  empty diamond head arrow and composition is denoted by filled diamond head arrow. When you draw UML diagram for two related class A and B, where A is associated with B then its denoted by A -> B. Similar way is used to show aggregation and composition between two classes. Here is UML notations for different kind of dependency between two classes.

[](http://1.bp.blogspot.com/-VL_9cjhwEE4/UvJN__IvaBI/AAAAAAAABCc/IkDmShgM-Yc/s1600/Association,+Composition+UML.JPG)

As, I said all three denotes relationship between object and only differ in their strength, you can also view them as below, where composition represent strongest form of relationship and association being the most general form.

[](http://3.bp.blogspot.com/-4KAOtX0S5NU/UvJN_w4TCZI/AAAAAAAABCo/SKPKYaci8s0/s1600/association+vs+composition+vs+aggregation.jpg)

Association vs Composition vs Aggregation

Here is the list of differences between Composition and Aggregation in point format, for quick review. As I said key difference between them comes from the point that in case of [Composition](http://javarevisited.blogspot.sg/2013/06/why-favor-composition-over-inheritance-java-oops-design.html), One object is OWNER of other object, while in case of aggregation, one object is just a USER or another object.  
  
1) If A and B two classes are related to each other such that, B ceased to exist, when A is destroyed, then association between two object is known as Composition. Example is Car and Engine. While if A and B are associated with each other, such that B can exist without being associated with A, then this association in known as Aggregation.  
  
2) In case of Composition A owns B e.g. Person is owner of his Hand, Mind and Heart, while  in case of Aggregation, A uses B e.g. Organization uses People as employee.  
  
3) In UML diagram Association is denoted by normal arrow head, while Composition is represented by filled diamond arrow head, and Aggregation is represented by empty diamond arrow head, As shown in below and attached diagram in third paragraph.  
  
Association  A---->B  
Composition  A-----<filled>B  
Aggregation  A-----<>B  
  
4) Aggregation is a lighter form of Composition, where sub-part object can meaningfully exits without main objects.  
  
5) In Java, you can use [final keyword](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) to represent Composition. Since in Composition, Owner object expect part object to be available and functions, by making it final, your provide guarantee that, when Owner will be created, this part object will exist. This is actually a *Java idiom to represent strong form of association* i.e. composition between two objects.  
  
6) Another interesting word, which comes handy to understand difference between Composition and Aggregation in software design is "part-of" and "has". If one object is part-of another object e.g. Engine is part of Car, then association or relationship between them is Composition. On the other hand if one object just has another object e.g. Car has driver than it's Aggregation.

**Association** is a relationship where all objects 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.

**Aggregation** is a specialised form of Association where all objects have their own lifecycle, but there is ownership and child objects can not belong to another parent object. Let’s take an example of Department and teacher. A single teacher can not belong to multiple departments, but if we delete the department teacher object will not be destroyed. We can think about it as a “has-a” relationship.

**Composition** is again specialised form of Aggregation and we can call this as a “death” relationship. It is a strong type of Aggregation. Child object does not have its lifecycle and if parent object is deleted, all child objects 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 belong to two different houses. If we delete the house - room will automatically be deleted. Let’s take another example relationship between Questions and Options. Single questions can have multiple options and option can not belong to multiple questions. If we delete questions options will automatically be deleted.

Understand the difference between inheritance and composition

|  |
| --- |
|  |

Stack extends Vector is a great example of where is-a wasn't really satisfied, and composition would have been much more flexible. Also, the Vector methods didn't really belong in Stack. When the retrofitted Vector to implement List, a whole bunch of other methods got added to the Stack interface.

A big part of the adapter pattern's significance is that it shows that objects involved in a composition relationship have interfaces that are independent of each other.

Ripple effect from changing a back-end class stops (or can stop) at the front-end class

## Composition vs. Inheritance

Inheritance Yields (Slightly) Better Performance:

* Composition's method forwarding/delegation will often be slower than inheritance's dynamic binding
* Composition results in more objects getting instantiated, which can incur a performance cost at allocation, <init>(), and GC time

Composition Yields Better Flexibility:

* Interfaces of classes involved in a composition relationship need not be compatible, so it's easier to change the interfaces
* Composition allows you to delay creation of back-end objects until (and unless) you need them
* Composition allows you to change back-end objects throughout the lifetime of the front-end object
* Composition allows front-end objects to share the same back-end objects

But:

* Composition's method forwarding/delegation results in more code that has to be written, debugged, and maintained.
* Easier to add new subclasses than new front-end classes, unless you use composition with interfaces

Describe adding method to superclass that has same name but different semantics as subclass. Also the problem of just inheriting an added method signature that you don't want in the subclass.

Class extension (full-blown inheritance -- interface and implementation) is like strapping on a backpack in which your instances carry around not only their own instance variables and everything they refer to in a graph of objects, but also those of all their super classes. If you don't own the superclass, owner can change superclass by adding lots of heavy instance data that gets added to your backpack and you don't have a choice but carry it around everywhere. The owner can also add a method to the superclass interface that you don't like in your subclass, but suddenly its a part of your interface whether you like it there or not (or, what's the ultimate problem has, the same signature but incompatible semantics as an existing method in your subclass). If the superclass owner adds lots of heavy instance data to the superclass, that owner basically has added all that weight to your subclass instance's backpack. Your subclass instances have no choice but carry that around everywhere they go. (In a composition relationship you could change the back-end object, delay initialization until needed, share same backend amount multiple front-ends, etc...)

## When to use abstract class and interface in Java

Here are some guidelines on when to use an abstract class and when to use interfaces in Java:

* An abstract class is good if you think you will plan on using inheritance since it provides a common base class implementation to derived classes.
* An abstract class is also good if you want to be able to declare non-public members. In an interface, all methods must be public.
* If you think you will need to add methods in the future, then an abstract class is a better choice. Because if you add new method headings to an interface, then all of the classes that already implement that interface will have to be changed to implement the new methods. That can be quite a hassle.
* Interfaces are a good choice when you think that the API will not change for a while.
* Interfaces are also good when you want to have something similar to multiple inheritance, since you can implement multiple interfaces.

Garbage Collection Algorithm

1. Reference Counting Collectors

Reference counting was an early garbage collection strategy. In this approach, a reference count is maintained for each object on the heap. When an object is first created and a reference to it is assigned to a variable, the object's reference count is set to one. When any other variable is assigned a reference to that object, the object's count is incremented. When a reference to an object goes out of scope or is assigned a new value, the object's count is decremented. Any object with a reference count of zero can be garbage collected. When an object is garbage collected, any objects that it refers to have their reference counts decremented. In this way the garbage collection of one object may lead to the subsequent garbage collection of other objects.

An advantage of this approach is that a reference counting collector can run in small chunks of time closely interwoven with the execution of the program. This characteristic makes it particularly suitable for real-time environments where the program can't be interrupted for very long. A disadvantage is that reference counting does not detect cycles: two or more objects that refer to one another. An example of a cycle is a parent object that has a reference to a child object that has a reference back to the parent. These objects will never have a reference count of zero even though they may be unreachable by the roots of the executing program. Another disadvantage of reference counting is the overhead of incrementing and decrementing the reference count each time.

Because of the disadvantages inherent in the reference counting approach, this technique is currently out of favor. It is more likely that the Java virtual machines you encounter in the real world will use a tracing algorithm in their garbage-collected heaps.

# 2. Tracing Collectors

Tracing garbage collectors trace out the graph of object references starting with the root nodes. Objects that are encountered during the trace are marked in some way. Marking is generally done by either setting flags in the objects themselves or by setting flags in a separate bitmap. After the trace is complete, unmarked objects are known to be unreachable and can be garbage collected.

The basic tracing algorithm is called "mark and sweep." This name refers to the two phases of the garbage collection process. In the mark phase, the garbage collector traverses the tree of references and marks each object it encounters. In the sweep phase, unmarked objects are freed, and the resulting memory is made available to the executing program. In the Java virtual machine, the sweep phase must include finalization of objects.

# Compacting Collectors

Garbage collectors of Java virtual machines will likely have a strategy to combat heap fragmentation. Two strategies commonly used by mark and sweep collectors are compacting and copying. Both of these approaches move objects on the fly to reduce heap fragmentation. Compacting collectors slide live objects over free memory space toward one end of the heap. In the process the other end of the heap becomes one large contiguous free area. All references to the moved objects are updated to refer to the new location.

Updating references to moved objects is sometimes made simpler by adding a level of indirection to object references. Instead of referring directly to objects on the heap, object references refer to a table of object handles. The object handles refer to the actual objects on the heap. When an object is moved, only the object handle must be updated with the new location. All references to the object in the executing program will still refer to the updated handle, which did not move. While this approach simplifies the job of heap defragmentation, it adds a performance overhead to every object access.

Copying Collectors

Copying garbage collectors move all live objects to a new area. As the objects are moved to the new area, they are placed side by side, thus eliminating any free space that may have separated them in the old area. The old area is then known to be all free space. The advantage of this approach is that objects can be copied as they are discovered by the traversal from the root nodes. There are no separate mark and sweep phases. Objects are copied to the new area on the fly, and forwarding pointers are left in their old locations. The forwarding pointers allow the garbage collector to detect references to objects that have already been moved. The garbage collector can then assign the value of the forwarding pointer to the references so they point to the object's new location.

A common copying collector algorithm is called "stop and copy." In this scheme, the heap is divided into two regions. Only one of the two regions is used at any time. Objects are allocated from one of the regions until all the space in that region has been exhausted. At that point program execution is stopped and the heap is traversed. Live objects are copied to the other region as they are encountered by the traversal. When the stop and copy procedure is finished, program execution resumes. Memory will be allocated from the new heap region until it too runs out of space. At that point the program will once again be stopped. The heap will be traversed and live objects will be copied back to the original region. The cost associated with this approach is that twice as much memory is needed for a given amount of heap space because only half of the available memory is used at any time.

You can see a graphical depiction of a garbage-collected heap that uses a stop and copy algorithm in Figure 9-1. This figure shows nine snapshots of the heap over time. In the first snapshot, the lower half of the heap is unused space. The upper half of the heap is partially filled by objects. That portion of the heap that contains objects is painted with diagonal gray lines. The second snapshot shows that the top half of the heap is gradually being filled up with objects, until it becomes full as shown in the third snapshot.

At that point, the garbage collector stops the program and traces out the graph of live objects starting with the root nodes. It copies each live object it encounters down to the bottom half of the heap, placing each object next to the previously copied object. This process is shown in snapshot four.

Snapshot five shows the heap after the garbage collection has finished. Now the top half of the heap is unused, and the bottom half is partially filled with live objects. The sixth snapshot shows the bottom half is now becoming gradually filled with objects, until it too becomes full in snapshot seven.

Once again, the garbage collector stops the program and traces out the graph of live objects. This time, it copies each live object it encounters up to the top half of the heap, as shown in snapshot eight. Snapshot nine shows the result of the garbage collection: the bottom half is once again unused space and the top half is partially filled with objects. This process repeats again and again as the program executes.

Finalization

In Java, an object may have a finalizer: a method that the garbage collector must run on the object prior to freeing the object. The potential existence of finalizers complicates the job of any garbage collector in a Java virtual machine.

To add a finalizer to a class, you simply declare a method in that class as follows:

// On CD-ROM in file gc/ex2/Example2.java

class Example2 {

protected void finalize() throws Throwable {

//...

super.finalize();

}

//...

}

A garbage collector must examine all objects it has discovered to be unreferenced to see if any include a finalize() method.

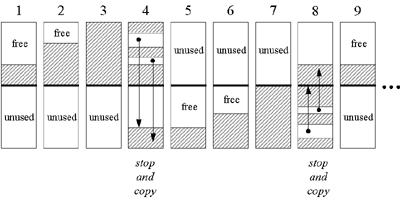
Because of finalizers, a garbage collector in the Java virtual machine must perform some extra steps each time it garbage collects. First, the garbage collector must in some way detect unreferenced objects (call this Pass I). Then, it must examine the unreferenced objects it has detected to see if any declare a finalizer. If it has enough time, it may at this point in the garbage collection process finalize all unreferenced objects that declare finalizers.

After executing all finalizers, the garbage collector must once again detect unreferenced objects starting with the root nodes (call this Pass II). This step is needed because finalizers can "resurrect" unreferenced objects and make them referenced again. Finally, the garbage collector can free all objects that were found to be unreferenced in both Passes I and II.

To reduce the time it takes to free up some memory, a garbage collector can optionally insert a step between the detection of unreferenced objects that have finalizers and the running of those finalizers. Once the garbage collector has performed Pass I and found the unreferenced objects that need to be finalized, it can run a miniature trace starting not with the root nodes but with the objects waiting to be finalized. Any objects that are (1) not reachable from the root nodes (those detected during Pass I) and (2) not reachable from the objects waiting to be finalized cannot be resurrected by any finalizer. These objects can be freed immediately.

If an object with a finalizer becomes unreferenced, and its finalizer is run, the garbage collector must in some way ensure that it never runs the finalizer on that object again. If that object is resurrected by its own finalizer or some other object's finalizer and later becomes unreferenced again, the garbage collector must treat it as an object that has no finalizer.

As you program in Java, you must keep in mind that it is the garbage collector that runs finalizers on objects. Because it is not generally possible to predict exactly when unreferenced objects will be garbage collected, it is not possible to predict when object finalizers will be run. As mentioned in Chapter 2, "Platform Independence," you should avoid writing programs for which correctness depends upon the timely finalization of objects. For example, if a finalizer of an unreferenced object releases a resource that is needed again later by the program, the resource will not be made available until after the garbage collector has run the object finalizer. If the program needs the resource before the garbage collector has gotten around to finalizing the unreferenced object, the program is out of luck.

  
  
**Figure 9-1. A "stop and copy" garbage-collected heap.**

# The Reachability Lifecycle of Objects

In versions prior to 1.2, every object on the heap is in one of three states from the perspective of the garbage collector: reachable, resurrectable, or unreachable. An object is in the reachablestate if the garbage collector can "reach" the object by tracing out the graph of object references starting with the root nodes. Every object begins its life in the reachable state, and stays reachable so long as the program maintains at least one reachable reference to the object. As soon as the program releases all references to an object, however, the object becomes resurrectable.

An object is in the resurrectable state if it is not currently reachable by tracing the graph of references starting with the root nodes, but could potentially be made reachable again later when the garbage collector executes some finalizer. All objects, not just objects that declare a finalize() method, pass through the resurrectable state. As mentioned in the previous section, the finalizer for an object may "resurrect" itself or any other resurrectable object by making the objects reachable again. Because any object in the resurrectable state could potentially be made reachable again by its own or some other object's finalize() method, the garbage collector cannot reclaim the memory occupied by a resurrectable object before it makes certain the object won't be brought back to life through the execution of a finalizer. By running the finalizers of all resurrectable objects that declare a finalize() method, the garbage collector will transform the state of all resurrectable objects, either back to the reachable state (for objects that get resurrected), or forward to the unreachable state.

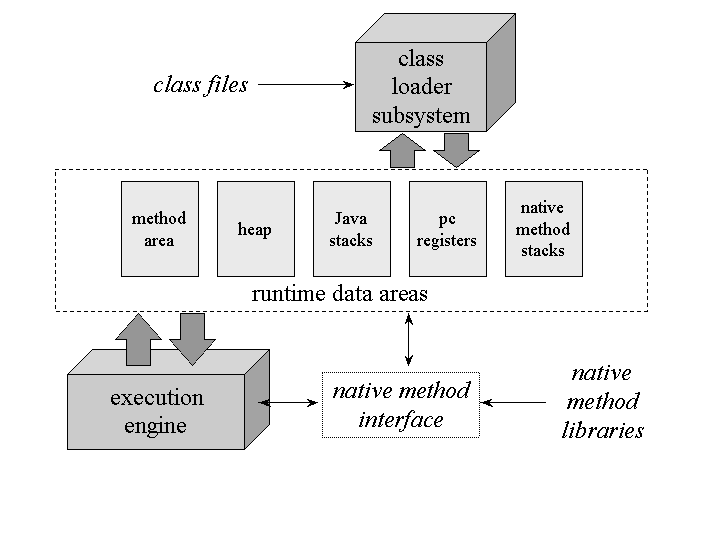
The unreachable state indicates not only that an object is no longer reachable, but also that the object cannot be made reachable again through the execution of some finalizer. Unreachable objects can no longer have any affect on the running program. The garbage collector, therefore, is free to reclaim the memory they occupy.

In version 1.2, the three original reachability states -- reachable, resurrectable, and unreachable -- were augmented by three new states: softly, weakly, and phantom reachable. Because these three new states represent three new (progressively weaker) kinds of reachability, the state known simply as "reachable" in versions prior to 1.2 is called "strongly reachable" starting with 1.2. Any object referenced directly from a root node, such as a local variable, is strongly reachable. Likewise, any object referenced directly from an instance variable of a strongly reachable object is strongly reachable.

The Architecture of the Java Virtual Machine

In the Java virtual machine specification, the behavior of a virtual machine instance is described in terms of subsystems, memory areas, data types, and instructions. These components describe an abstract inner architecture for the abstract Java virtual machine. The purpose of these components is not so much to dictate an inner architecture for implementations. It is more to provide a way to strictly define the external behavior of implementations. The specification defines the required behavior of any Java virtual machine implementation in terms of these abstract components and their interactions.

Figure 5-1 shows a block diagram of the Java virtual machine that includes the major subsystems and memory areas described in the specification. As mentioned in previous chapters, each Java virtual machine has a *class loader subsystem*: a mechanism for loading types (classes and interfaces) given fully qualified names. Each Java virtual machine also has an *execution engine*: a mechanism responsible for executing the instructions contained in the methods of loaded classes.

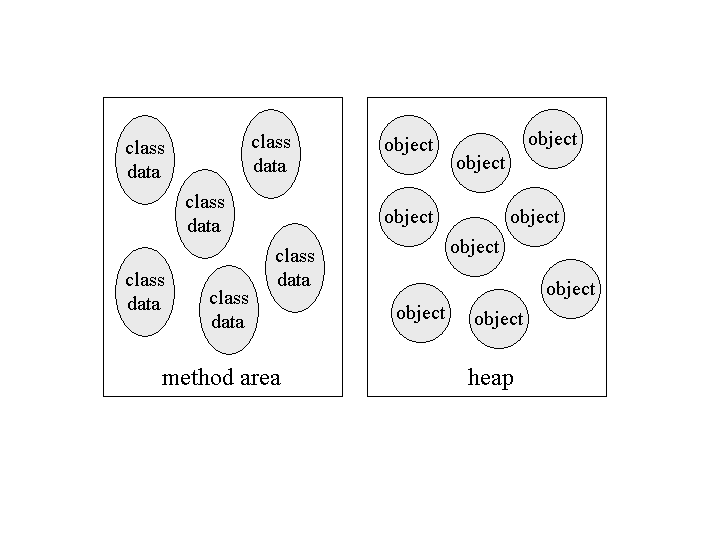
  
  
**Figure 5-1. The internal architecture of the Java virtual machine.**

When a Java virtual machine runs a program, it needs memory to store many things, including bytecodes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations. The Java virtual machine organizes the memory it needs to execute a program into several *runtime data areas*.

Although the same runtime data areas exist in some form in every Java virtual machine implementation, their specification is quite abstract. Many decisions about the structural details of the runtime data areas are left to the designers of individual implementations.

Different implementations of the virtual machine can have very different memory constraints. Some implementations may have a lot of memory in which to work, others may have very little. Some implementations may be able to take advantage of virtual memory, others may not. The abstract nature of the specification of the runtime data areas helps make it easier to implement the Java virtual machine on a wide variety of computers and devices.

Some runtime data areas are shared among all of an application's threads and others are unique to individual threads. Each instance of the Java virtual machine has one *method area* and one*heap*. These areas are shared by all threads running inside the virtual machine. When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area. As the program runs, the virtual machine places all objects the program instantiates onto the heap. See Figure 5-2 for a graphical depiction of these memory areas.



**Figure 5-2. Runtime data areas shared among all threads.**

As each new thread comes into existence, it gets its own *pc register* (program counter) and *Java stack*. If the thread is executing a Java method (not a native method), the value of the pc register indicates the next instruction to execute. A thread's Java stack stores the state of Java (not native) method invocations for the thread. The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations. The state of native method invocations is stored in an implementation-dependent way in *native method stacks*, as well as possibly in registers or other implementation-dependent memory areas.

The Java stack is composed of *stack frames* (or *frames*). A stack frame contains the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual machine pops and discards the frame for that method.

The Java virtual machine has no registers to hold intermediate data values. The instruction set uses the Java stack for storage of intermediate data values. This approach was taken by Java's designers to keep the Java virtual machine's instruction set compact and to facilitate implementation on architectures with few or irregular general purpose registers. In addition, the stack-based architecture of the Java virtual machine's instruction set facilitates the code optimization work done by just-in-time and dynamic compilers that operate at run-time in some virtual machine implementations.

See Figure 5-3 for a graphical depiction of the memory areas the Java virtual machine creates for each thread. These areas are private to the owning thread. No thread can access the pc register or Java stack of another thread.

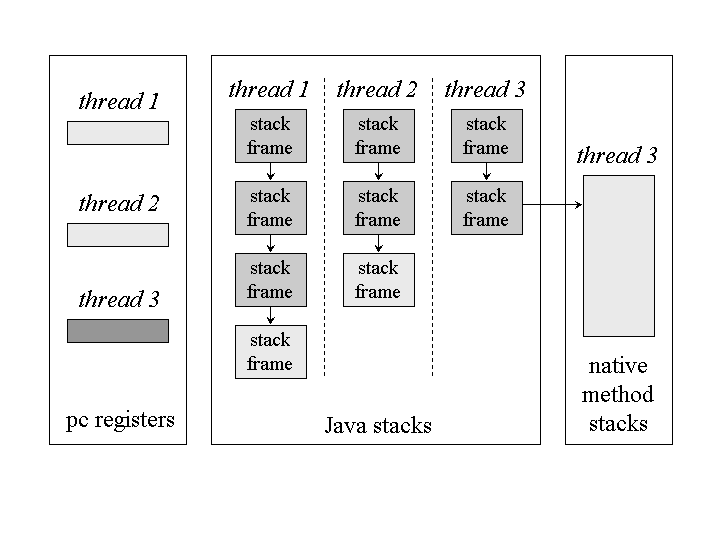
  
  
**Figure 5-3. Runtime data areas exclusive to each thread.**

Figure 5-3 shows a snapshot of a virtual machine instance in which three threads are executing. At the instant of the snapshot, threads one and two are executing Java methods. Thread three is executing a native method.

In Figure 5-3, as in all graphical depictions of the Java stack in this book, the stacks are shown growing downwards. The "top" of each stack is shown at the bottom of the figure. Stack frames for currently executing methods are shown in a lighter shade. For threads that are currently executing a Java method, the pc register indicates the next instruction to execute. In Figure 5-3, such pc registers (the ones for threads one and two) are shown in a lighter shade. Because thread three is currently executing a native method, the contents of its pc register--the one shown in dark gray--is undefined

## The Class Loader Subsystem

The part of a Java virtual machine implementation that takes care of finding and loading types is the class loader subsystem. Chapter 1, "Introduction to Java's Architecture," gives an overview of this subsystem. Chapter 3, "Security," shows how the subsystem fits into Java's security model. This chapter describes the class loader subsystem in more detail and show how it relates to the other components of the virtual machine's internal architecture.

As mentioned in Chapter 1, the Java virtual machine contains two kinds of class loaders: a bootstrap class loader and user-defined class loaders. The bootstrap class loader is a part of the virtual machine implementation, and user-defined class loaders are part of the running Java application. Classes loaded by different class loaders are placed into separate name spaces inside the Java virtual machine.

The class loader subsystem involves many other parts of the Java virtual machine and several classes from the java.lang library. For example, user-defined class loaders are regular Java objects whose class descends from java.lang.ClassLoader. The methods of class ClassLoader allow Java applications to access the virtual machine's class loading machinery. Also, for every type a Java virtual machine loads, it creates an instance of class java.lang.Class to represent that type. Like all objects, user-defined class loaders and instances of class Class reside on the heap. Data for loaded types resides in the method area.

### Loading, Linking and Initialization

The class loader subsystem is responsible for more than just locating and importing the binary data for classes. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references. These activities are performed in a strict order:

1. Loading: finding and importing the binary data for a type
2. Linking: performing verification, preparation, and (optionally) resolution
   1. Verification: ensuring the correctness of the imported type
   2. Preparation: allocating memory for class variables and initializing the memory to default values
   3. Resolution: transforming symbolic references from the type into direct references.
3. Initialization: invoking Java code that initializes class variables to their proper starting values.

The details of these processes are given Chapter 7, "The Lifetime of a Type."

### The Bootstrap Class Loader

Java virtual machine implementations must be able to recognize and load classes and interfaces stored in binary files that conform to the Java class file format. An implementation is free to recognize other binary forms besides class files, but it must recognize class files.

Every Java virtual machine implementation has a bootstrap class loader, which knows how to load trusted classes, including the classes of the Java API. The Java virtual machine specification doesn't define how the bootstrap loader should locate classes. That is another decision the specification leaves to implementation designers.

Given a fully qualified type name, the bootstrap class loader must in some way attempt to produce the data that defines the type. One common approach is demonstrated by the Java virtual machine implementation in Sun's 1.1 JDK on Windows98. This implementation searches a user-defined directory path stored in an environment variable named CLASSPATH. The bootstrap loader looks in each directory, in the order the directories appear in the CLASSPATH, until it finds a file with the appropriate name: the type's simple name plus ".class". Unless the type is part of the unnamed package, the bootstrap loader expects the file to be in a subdirectory of one the directories in the CLASSPATH. The path name of the subdirectory is built from the package name of the type. For example, if the bootstrap class loader is searching for class java.lang.Object, it will look for Object.class in the java\lang subdirectory of each CLASSPATH directory.

In 1.2, the bootstrap class loader of Sun's Java 2 SDK only looks in the directory in which the system classes (the class files of the Java API) were installed. The bootstrap class loader of the implementation of the Java virtual machine from Sun's Java 2 SDK does not look on the CLASSPATH. In Sun's Java 2 SDK virtual machine, searching the class path is the job of the system class loader, a user-defined class loader that is created automatically when the virtual machine starts up. More information on the class loading scheme of Sun's Java 2 SDK is given in Chapter 8, "The Linking Model."

### User-Defined Class Loaders

Although user-defined class loaders themselves are part of the Java application, four of the methods in class ClassLoader are gateways into the Java virtual machine:

// Four of the methods declared in class java.lang.ClassLoader:

protected final Class defineClass(String name, byte data[],

int offset, int length);

protected final Class defineClass(String name, byte data[],

int offset, int length, ProtectionDomain protectionDomain);

protected final Class findSystemClass(String name);

protected final void resolveClass(Class c);

Any Java virtual machine implementation must take care to connect these methods of class ClassLoader to the internal class loader subsystem.

The two overloaded defineClass() methods accept a byte array, data[], as input. Starting at position offset in the array and continuing for length bytes, class ClassLoader expects binary data conforming to the Java class file format--binary data that represents a new type for the running application -- with the fully qualified name specified in name. The type is assigned to either a default protection domain, if the first version of defineClass() is used, or to the protection domain object referenced by the protectionDomain parameter. Every Java virtual machine implementation must make sure the defineClass() method of class ClassLoader can cause a new type to be imported into the method area.

The findSystemClass() method accepts a String representing a fully qualified name of a type. When a user-defined class loader invokes this method in version 1.0 and 1.1, it is requesting that the virtual machine attempt to load the named type via its bootstrap class loader. If the bootstrap class loader has already loaded or successfully loads the type, it returns a reference to theClass object representing the type. If it can't locate the binary data for the type, it throws ClassNotFoundException. In version 1.2, the findSystemClass() method attempts to load the requested type from the system class loader. Every Java virtual machine implementation must make sure the findSystemClass() method can invoke the bootstrap (if version 1.0 or 1.1) or system (if version 1.2 or later) class loader in this way.

The resolveClass() method accepts a reference to a Class instance. This method causes the type represented by the Class instance to be linked (if it hasn't already been linked). ThedefineClass() method, described previous, only takes care of loading. (See the previous section, "Loading, Linking, and Initialization" for definitions of these terms.) When defineClass()returns a Class instance, the binary file for the type has definitely been located and imported into the method area, but not necessarily linked and initialized. Java virtual machine implementations make sure the resolveClass() method of class ClassLoader can cause the class loader subsystem to perform linking.

The details of how a Java virtual machine performs class loading, linking, and initialization, with user- defined class loaders is given in Chapter 8, "The Linking Model."

**Name Spaces**

As mentioned in Chapter 3, "Security," each class loader maintains its own name space populated by the types it has loaded. Because each class loader has its own name space, a single Java application can load multiple types with the same fully qualified name. A type's fully qualified name, therefore, is not always enough to uniquely identify it inside a Java virtual machine instance. If multiple types of that same name have been loaded into different name spaces, the identity of the class loader that loaded the type (the identity of the name space it is in) will also be needed to uniquely identify that type.

Name spaces arise inside a Java virtual machine instance as a result of the process of resolution. As part of the data for each loaded type, the Java virtual machine keeps track of the class loader that imported the type. When the virtual machine needs to resolve a symbolic reference from one class to another, it requests the referenced class from the same class loader that loaded the referencing class. This process is described in detail in Chapter 8, "The Linking Model."