Stack Frame

Java stores stuff in two separate pools of memory: the stack and the heap. The heap stores all objects, including all arrays, and all class variables (i.e. those declared "static"). The stack stores all local variables, including all parameters.
When a method is called, the Java Virtual Machine creates a stack frame (also known as an activation record) that stores the parameters and local variables for that method. One method can call another, which can call another, and so on, so the JVM maintains an internal stack of stack frames, with "main" at the bottom, and the most recent method call on top.

The Stack Frame

The stack frame has three parts**: local variables, operand stack, and frame data**. The sizes of the local variables and operand stack, which are measured in words, depend upon the needs of each individual method. These sizes are determined at compile time and included in the class file data for each method. The size of the frame data is implementation dependent.

Local Variables

The local variables section of the Java stack frame is organized as a zero-based array of words. Instructions that use a value from the local variables section provide an index into the zero-based array. Values of type int, float, reference, and return Address occupy one entry in the local variables array. Values of type byte, short, and char are converted to int before being stored into the local variables. Values of type long and double occupy two consecutive entries in the array. **Figure 1 shows the local variables section for the following two methods:**

```
class Example3a {
   public static int runClassMethod(int i, long l, float f, double d, Object o, byte b) {
      return 0;
   }
   public int runInstanceMethod(char c, double d, short s, boolean b) {
      return 0;
   }
}
```

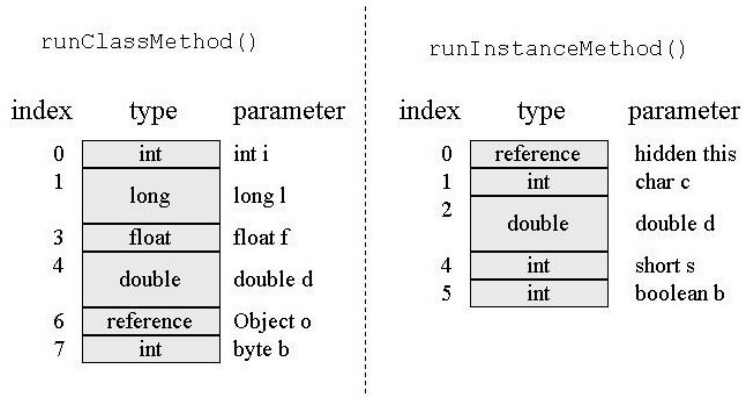


Figure 1. Methods parameters on the local variables section of a Java stack.

When a method finishes executing, its stack frame is erased from the top of the stack, and its local variables are erased forever.
The java.lang library has a method "Thread.dumpStack" that prints a list of the methods on the stack (but it doesn't print their local variables). This method can be convenient for debugging--for instance, when you're trying to figure out which method called another method with illegal parameters that made it crash.

Parameter Passing
As in Scheme, Java passes all parameters by value. This means that the method has copies of the actual parameters, and cannot change the originals. The copies reside in the method's stack frame for the method. The method can change these copies, but the original values that were copied are not changed.
In this example, the method doNothing sets its parameter to 2, but it has no effect on the value of the calling method's variable a:

```
method:
static void doNothing(int x) {
    x = 2;
  }
method call:

int a = 1;
doNothing(a);
```

```
STACK (just before method returns)
  -----
x | 2 |
  -----     stack frame for doNothing
-------------------------------------
---
  -----
a | 1 |
  -----     stack frame for main
```

When the method call returns, **a** is still 1. The doNothing method, as its name suggests, failed to change the value of **a**, or do anything relevant at all.

However, when a parameter is a reference to an object, the reference is copied, but the object is not; the original object is shared. A method can modify an object that one of its parameters points to, and the change will be visible everywhere. Here's an example that shows how a method can make a change to an object that is visible to the calling method:

```
method:

class IntBox {
  public int i;
  static void set3(IntBox ib) {
    ib.i = 3;
  }
method call:
  IntBox b = new IntBox();
  set3(b);
```

```
STACK              | HEAP
                set3|
       -----        |
   ib | .-+--------------|
       -----        |      |
                    |      |
 ------------------ |      |
       -----          |    ------
    b | .-+------------->|i |3|
       -----     main|    ------
```

For those of you who are familiar with programming languages that have "pass by reference," the example above is as close as you can get in Java. But it's not "pass by reference." Rather, it's passing a reference by value.

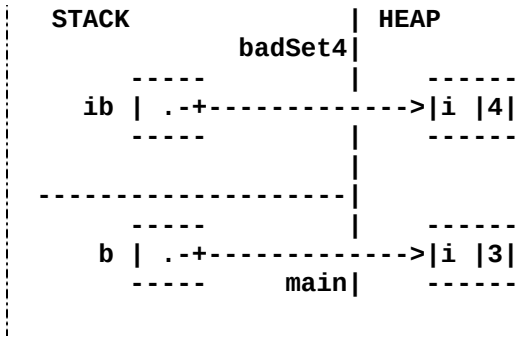
Here's an example of a common programming error, where a method tries and fails to make a change that is visible to the calling method. (Assume we've just executed the example above, so b is set up.)

```
method:

class IntBox {
  static void badSet4(IntBox ib) {
  ib = new IntBox();
  ib.i = 4;
}

method call:

badSet4(b);
```



JAVA PACKAGES

In Java, a package is a collection of classes and Java interfaces. Packages have **three benefits**.

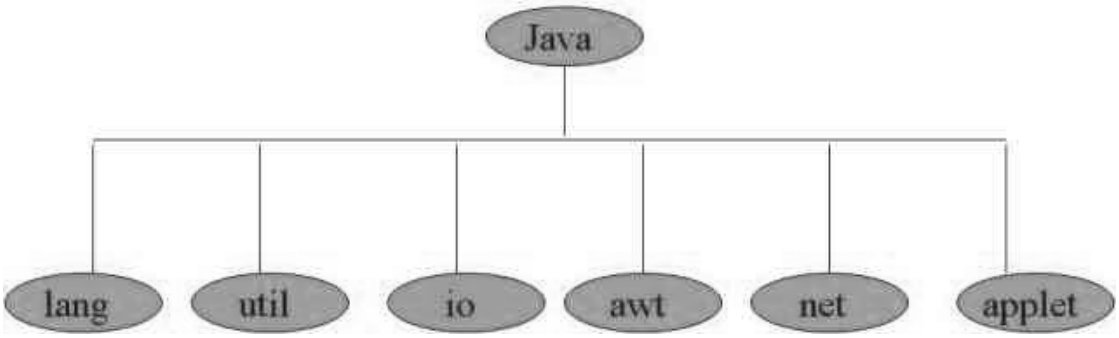
(1) Packages can contain hidden classes that are used by the package but are not visible or accessible outside the package.
(2) Classes in packages can have fields and methods that are visible by all classes inside the package, but not outside.
(3) Different packages can have classes with the same name. For example, java.awt.Frame and photo.Frame.

Here are **two examples of packages**.

(1) java.io is a package of I/O-related classes in the standard Java libraries.
(2) "list", a package containing the classes DList and DListNode. You will be adding two additional classes to the list package.

Package names are hierarchical. java.awt.image. Model refers to the class Model inside the package image inside the package awt inside the package java.

Java API Packages



java.lang: Language support classes. These are the classes that Java compiler itself uses and therefore they are automatically imported. They include classes of primitive types, strings, math functions, threads and exceptions.
java.util: Language utility classes such as vector, hash tables, random numbers, data etc.
java.io: Input/output support classes. They provide facilities for the input and output of data
java.awt: set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
java.net: Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.applet: Classes for creating and implementing applets.

Package java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenize, a random-number generator, and a bit array).

Interface Summary	
Collection	The root interface in the collection hierarchy.
Comparator	A comparison function, which imposes a total ordering on some collection of objects.
Enumeration	An object that implements the Enumeration interface generates a series of elements, one at a time.
EventListener	A tagging interface that all event listener interfaces must extend.
Iterator	An iterator over a collection.
List	An ordered collection (also known as a sequence).
ListIterator	An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
Map	An object that maps keys to values.
Map.Entry	A map entry (key-value pair).
Observer	A class can implement the Observer interface when it wants to be informed of changes in observable objects.
RandomAccess	Marker interface used by List implementations to indicate that they support fast (generally constant time) random access.
Set	A collection that contains no duplicate elements.
SortedMap	A map that further guarantees that it will be in ascending key order, sorted according to the natural ordering of its keys (see the Comparable interface), or by a comparator provided at sorted map creation time.
SortedSet	A set that further guarantees that its iterator will traverse the set in ascending

	element order, sorted according to the natural ordering of its elements (see Comparable), or by a Comparator provided at sorted set creation time.