1

const int x; // x is a constant data variable

int const x; // x is a constant data variable

const int\* x; // x is a non-constant pointer to constant data

int const\* x; // x is a non-constant pointer to constant data

int\*const x; // x is a constant pointer to non-constant data

const int\*const x; // x is a constant pointer to constant data

And finally, it is possible to combine different type qualifiers. For example a volatile const int\* is a non-constant, non-volatile pointer to volatile const data. Once again we have the wonderful option to mix the order of these to confuse, so we can also write volatile int const \* or int volatile const \* etc and it all means the same.

2

#define REG(x) (\*((volatile unsigned int \*)(x)))

const int\*const x; means a constant pointer pointing at a constant int.

3

TEST

#DEFINE BASE (unsigned cons int\*)0x0038

volatile unsigned cons int\* ptr;

\*ptr = BASE;

set\_bit(5,\*ptr);

**synchronization**

http://www.makelinux.net/books/lkd2/ch09lev1sec2

**1. Atomic variable**

it is never possible for the two atomic operations to occur on the same variable concurrently

Atomic Integer Operations

he atomic integer methods operate on a special data type, atomic\_t.

Defining an atomic\_t is done in the usual manner. Optionally, you can set it to an initial value:

atomic\_t v; /\* define v \*/

atomic\_t u = ATOMIC\_INIT(0); /\* define u and initialize it to zero \*/

Operations are all simple:

atomic\_set(&v, 4); /\* v = 4 (atomically) \*/

atomic\_add(2, &v); /\* v = v + 2 = 6 (atomically) \*/

atomic\_inc(&v); /\* v = v + 1 = 7 (atomically) \*/

If you ever need to convert an atomic\_t to an int, use atomic\_read():

printk("%d\n", atomic\_read(&v)); /\* will print "7" \*/

A common use of the atomic integer operations is to implement counters. Protecting a sole counter with a complex locking scheme is silly, so instead developers use atomic\_inc() and atomic\_dec(), which are much lighter in weight.

Another use of the atomic integer operators is atomically performing an operation and testing the result. A common example is the atomic decrement and test:

int atomic\_dec\_and\_test(atomic\_t \*v)

This function decrements by one the given atomic value. If the result is zero, it returns true; otherwise, it returns false. A full listing of the standard atomic integer operations (those found on all architectures) is in Table 9.1. All the operations implemented on a specific architecture can be found in <asm/atomic.h>.

Atomic Integer Operation

Description

ATOMIC\_INIT(int i)

At declaration, initialize an atomic\_t to i

int atomic\_read(atomic\_t \*v)

Atomically read the integer value of v

void atomic\_set(atomic\_t \*v, int i)

Atomically set v equal to i

void atomic\_add(int i, atomic\_t \*v)

Atomically add i to v

void atomic\_sub(int i, atomic\_t \*v)

Atomically subtract i from v

void atomic\_inc(atomic\_t \*v)

Atomically add one to v

void atomic\_dec(atomic\_t \*v)

Atomically subtract one from v

int atomic\_sub\_and\_test(int i, atomic\_t \*v)

Atomically subtract i from v and return true if the result is zero; otherwise false

int atomic\_add\_negative(int i, atomic\_t \*v)

Atomically add i to v and return true if the result is negative; otherwise false

int atomic\_dec\_and\_test(atomic\_t \*v)

Atomically decrement v by one and return true if zero; false otherwise

int atomic\_inc\_and\_test(atomic\_t \*v)

Atomically increment v by one and return true if the result is zero; false otherwise

Atomic Bitwise Operations

In addition to atomic integer operations, the kernel also provides a family of functions that operate at the bit level. Not surprisingly, they are architecture specific and defined in <asm/bitops.h>.

For example, assume you issue two atomic bit operations: Initially set the bit and then clear the bit. Without atomic operations, the bit may end up cleared, but it may never have been set. The set operation could occur simultaneously with the clear operation and fail. The clear operation would succeed, and the bit would emerge cleared as intended. With atomic operations, however, the set would actually occurthere would be a moment in time when a read would show the bit as setand then the clear would execute and the bit be zero.

**2 Spin Locks**

Although it would be nice if every critical region consisted of code that did nothing more complicated than incrementing a variable, reality is much crueler. In real life, critical regions can span multiple functions. For example, it is often the case that data must be removed from one structure, formatted and parsed, and added to another structure. This entire operation must occur atomically; it must not be possible for other code to read from or write to either structure before its update is done. Because simple atomic operations are clearly incapable of providing the needed protection in such a complex scenario, a more general method of synchronization is needed.

The most common lock in the Linux kernel is the spin lock. A spin lock is a lock that can be held by at most one thread of execution. If a thread of execution attempts to acquire a spin lock while it is contended (already held), the thread busy loop spins waiting for the lock to become available. If the lock is not contended, the thread can immediately acquire the lock and continue. The spinning prevents more than one thread of execution from entering the critical region at any one time. Note that the same lock can be used in multiple locations, so all access to a given data structure, for example, can be protected and synchronized.

The fact that a contended spin lock causes threads to spin (essentially wasting processor time) while waiting for the lock to become available is important. This behavior is the point of the spin lock. It is not wise to hold a spin lock for a long time. This is the nature of the spin lock: a lightweight single-holder lock that should be held for short durations. An alternative behavior when the lock is contended is to put the current thread to sleep and wake it up when it becomes available. Then the processor can go off and execute other code.

The next section covers semaphores, which provide a lock that makes the waiting thread sleep, rather than spin, when contended.

Spin locks are architecture dependent and implemented in assembly. The architecture-dependent code is defined in <asm/spinlock.h>. The actual usable interfaces are defined in <linux/spinlock.h>. The basic use of a spin lock is

spinlock\_t mr\_lock = SPIN\_LOCK\_UNLOCKED;

spin\_lock(&mr\_lock);

/\* critical region \*/

spin\_unlock(&mr\_lock);

Warning: if you attempt to acquire a lock you already hold, you will spin, waiting for yourself to release the lock. But because you are busy spinning, you will never release the lock and you will deadlock. Be careful!

**Using spinlocks in interrupt handlers.**

Spin locks can be used in interrupt handlers, whereas semaphores cannot be used because they sleep. If a lock is used in an interrupt handler, you must also disable local interrupts (interrupt requests on the current processor) before obtaining the lock. Otherwise, it is possible for an interrupt handler to interrupt kernel code while the lock is held and attempt to reacquire the lock. The interrupt handler spins, waiting for the lock to become available. The lock holder, however, does not run until the interrupt handler completes. This is an example of the double-acquire deadlock.

ote that you need to disable interrupts only on the current processor. If an interrupt occurs on a different processor, and it spins on the same lock, it does not prevent the lock holder (which is on a different processor) from eventually releasing the lock.

The kernel provides an interface that conveniently disables interrupts and acquires the lock. Usage is

spinlock\_t mr\_lock = SPIN\_LOCK\_UNLOCKED;

unsigned long flags;

spin\_lock\_irqsave(&mr\_lock, flags);

/\* critical region ... \*/

spin\_unlock\_irqrestore(&mr\_lock, flags);

The routine spin\_lock\_irqsave() saves the current state of interrupts, disables them locally, and then obtains the given lock. Conversely, spin\_unlock\_irqrestore() unlocks the given lock and returns interrupts to their previous state. This way, if interrupts were initially disabled, your code would not erroneously enable them, but instead keep them disabled.

If you always know before the fact that interrupts are initially enabled, there is no need to restore their previous state. You can unconditionally enable them on unlock. In those cases, spin\_lock\_irq() and spin\_unlock\_irq() are optimal:

spinlock\_t mr\_lock = SPIN\_LOCK\_UNLOCKED;

spin\_lock\_irq(&mr\_lock);

/\* critical section ... \*/

spin\_unlock\_irq(&mr\_lock);

Debugging Spin Locks

The configure option CONFIG\_DEBUG\_SPINLOCK enables a handful of debugging checks in the spin lock code.

**Other Spin Lock Methods**

The method spin\_trylock() attempts to obtain the given spin lock. If the lock is contended, rather than spin and wait for the lock to be released, the function immedi-ately returns zero. If it succeeds in obtaining the lock, it returns nonzero. Similarly, spin\_is\_locked() returns nonzero if the given lock is currently acquired. Otherwise, it returns zero. In neither case does this function actually obtain the lock.

**Reader-Writer Spin Locks**

Sometimes, lock usage can be clearly divided into readers and writers. For example, consider a list that is both updated and searched. When the list is updated (written to), it is important that no other threads of execution concurrently write to or read from the list. Writing demands mutual exclusion. On the other hand, when the list is searched (read from), it is only important that nothing else write to the list. Multiple concurrent readers are safe so long as there are no writers.

When a data structure is neatly split into reader/writer paths like this, it makes sense to use a locking mechanism that provides similar semantics. In this case, Linux provides reader-writer spin locks. Reader-writer spin locks provide separate reader and writer variants of the lock. One or more readers can concurrently hold the reader lock. The writer lock, conversely, can be held by at most one writer with no concurrent readers. Reader/writer locks are sometimes called shared/exclusive or concurrent/exclusive locks because the lock is available in a shared (for readers) and an exclusive (for writers) form.

Usage is similar to spin locks. The reader-writer spin lock is initialized via

rwlock\_t mr\_rwlock = RW\_LOCK\_UNLOCKED;

Then, in the reader code path:

read\_lock(&mr\_rwlock);

/\* critical section (read only) ... \*/

read\_unlock(&mr\_rwlock);

Finally, in the writer code path:

write\_lock(&mr\_rwlock);

/\* critical section (read and write) ... \*/

write\_unlock(&mr\_lock);

Note

If the line between your readers and writers is muddled, it might be an indication that you do not need to use reader-writer locks. In that case, a normal spin lock is optimal.

A final important consideration in using the Linux reader-writer spin locks is that they favour readers over writers. If the read lock is held and a writer is waiting for exclusive access, readers that attempt to acquire the lock will continue to succeed. The spinning writer does not acquire the lock until all readers release the lock. Therefore, a sufficient number of readers can starve pending writers. This is important to keep in mind when designing your locking.

**3. Semaphores**

Semaphores in Linux are sleeping locks. When a task attempts to acquire a semaphore that is already held, the semaphore places the task onto a wait queue and puts the task to sleep. The processor is then free to execute other code. When the processes[3] holding the semaphore release the lock, one of the tasks on the wait queue is awakened so that it can then acquire the semaphore.

Let's jump back to the door and key analogy. When a person reaches the door, he can grab the key and enter the room. The big difference lies in what happens when another dude reaches the door and the key is not available. In this case, instead of spinning, the fellow puts his name on a list and takes a nap. When the person inside the room leaves, he checks the list at the door. If anyone's name is on the list, he goes over to the first name and gives him a playful jab in the chest, waking him up and allowing him to enter the room. In this manner, the key (read: semaphore) continues to ensure that there is only one person (read: thread of execution) inside the room (read: critical region) at one time. If the room is occupied, instead of spinning, the person puts his name on a list (read: wait queue) and takes a nap (read: blocks on the wait queue and goes to sleep), allowing the processor to go off and execute other code. This provides better processor utilization than spin locks because there is no time spent busy looping, but semaphores have much greater overhead than spin locks. Life is always a trade-off.

You can draw some interesting conclusions from the sleeping behavior of semaphores:

1. Because the contending tasks sleep while waiting for the lock to become available, semaphores are well suited to locks that are held for a long time.

2. Conversely, semaphores are not optimal for locks that are held for very short periods because the overhead of sleeping, maintaining the wait queue, and waking back up can easily outweigh the total lock hold time.

3. You can (although you may not want to) sleep while holding a semaphore because you will not deadlock when another process acquires the same semaphore.

4. You cannot hold a spin lock while you acquire a semaphore, because you might have to sleep while waiting for the semaphore, and you cannot sleep while holding a spin lock.

Whereas spin locks permit at most one task to hold the lock at a time, the number of permissible simultaneous holders of semaphores can be set at declaration time. This value is called the usage count or simply the count. The most common value is to allow, like spin locks, only one lock holder at a time. In this case, the count is equal to one and the semaphore is called either a binary semaphore (because it is either held by one task or not held at all) or a mutex (because it enforces mutual exclusion).

Alternatively, the count can be initialized to a nonzero value greater than one. In this case, the semaphore is called a counting semaphore, and it allows at most count holders of the lock at a time. Counting semaphores are not used to enforce mutual exclusion because they allow multiple threads of execution in the critical region at once. Instead, they are used to enforce limits in certain code.

**Using Semaphores**

The function down\_interruptible() attempts to acquire the given semaphore. If it fails, it sleeps in the TASK\_INTERRUPTIBLE state. Recall from Chapter 3 that this process state implies that a task can be awakened with a signal, which is generally a good thing.

If the task receives a signal while waiting for the semaphore, it is awakened and down\_interruptible() returns -EINTR

You can use down\_trylock() to try to acquire the given semaphore with blocking. If the semaphore is already held, the function immediately returns nonzero. Otherwise, it returns zero and you successfully hold the lock.

To release a given semaphore, call up(). Consider an example:

/\* define and declare a semaphore, named mr\_sem, with a count of one \*/

static DECLARE\_MUTEX(mr\_sem);

/\* attempt to acquire the semaphore ... \*/

if (down\_interruptible(&mr\_sem)) {

/\* signal received, semaphore not acquired ... \*/

}

/\* critical region ... \*/

/\* release the given semaphore \*/

up(&mr\_sem);

Table 9.6. What to Use: Spin Locks Versus Semaphores

Requirement Recommended Lock

Low overhead locking Spin lock is preferred

Short lock hold time Spin lock is preferred

Long lock hold time Semaphore is preferred

Need to lock from interrupt context Spin lock is required

Need to sleep while holding lock Semaphore is required

**Chapter 10. Timers and Time Management**

The passing of time is very important to the kernel. A large number of kernel functions are time driven, as opposed to event driven[1]. Some of these functions are periodic, such as balancing the scheduler run queues or refreshing the screen. They occur on a fixed schedule, such as 100 times per second.

Events that occur periodically say, every 10 milliseconds are driven by the system timer. The system timer is a programmable piece of hardware that issues an interrupt at a fixed frequency. The interrupt handler for this timer called the timer interrupt updates the system time and performs periodic work.

The other focus is dynamic timers the facility used to schedule events that run once after a specified time has elapsed. For example, the floppy device driver uses a timer to shut off the floppy drive motor after a specified period of inactivity. The kernel can create and destroy timers dynamically.

The Tick Rate: HZ

The frequency of the system timer (the tick rate) is programmed on system boot based on a static preprocessor define, HZ. The value of HZ differs for each supported architecture. In fact, on some supported architectures, it even differs between machine types.

The kernel defines the value in <asm/param.h>. The tick rate has a frequency of HZ hertz and a period of 1/HZ seconds. For example, in include/asm-i386/param.h, the i386 architecture defines:

#define HZ 1000 /\* internal kernel time frequency \*/

**Jiffies**

The global variable jiffies holds the number of ticks that have occurred since the system booted. On boot, the kernel initializes the variable to zero, and it is incremented by one during each timer interrupt. Thus, because there are HZ timer interrupts in a second, there are HZ jiffies in a second. The system uptime is therefore jiffies/HZ seconds.

The jiffies variable is declared in <linux/jiffies.h> as

extern unsigned long volatile jiffies;

In the next section, we will look at its actual definition, which is a bit peculiar. For now, let's look at some sample kernel code. The following code converts from seconds to a unit of jiffies:

(seconds \* HZ)

Likewise, this code converts from jiffies to seconds:

(jiffies / HZ)

for example:

unsigned long time\_stamp = jiffies; /\* now \*/

unsigned long next\_tick = jiffies +\_1; /\* one tick from now \*/

unsigned long later = jiffies + 5\*HZ; /\* five seconds from now \*/

Internal Representation of Jiffies

The jiffies variable has always been an unsigned long, and therefore 32 bits in size on 32-bit architectures and 64-bits on 64-bit architectures.

With a tick rate of 100, a 32-bit jiffies variable would overflow in about 497 days. With HZ increased to 1000, however, that overflow now occurs in just 49.7 days! If jiffies were stored in a 64-bit variable on all architectures, then for any reasonable HZ value the jiffies variable would never overflow in anyone's lifetime.

As you previously saw, jiffies is defined as an unsigned long:

extern unsigned long volatile jiffies;

A second variable is also defined in <linux/jiffies.h>:

extern u64 jiffies\_64;

The ld(1) script used to link the main kernel image (arch/i386/kernel/vmlinux.lds.S on x86) then overlays the jiffies variable over the start of the jiffies\_64 variable:

jiffies = jiffies\_64;

Thus, jiffies is the lower 32 bits of the full 64-bit jiffies\_64 variable. Code can continue to access the jiffies variable exactly as before. Because most code uses jiffies simply to measure elapses in time, most code cares about only the lower 32 bits.

Code that accesses jiffies simply reads the lower 32 bits of jiffies\_64. The function get\_jiffies\_64() can be used to read the full 64-bit value[5]. Such a need is rare; consequently, most code simply continues to read the lower 32 bits directly via the jiffies variable.

On 64-bit architectures, jiffies\_64 and jiffies refer to the same thing. Code can either read jiffies or call get\_jiffies\_64() because both actions have the same effect.

Look at an example of a wraparound:

unsigned long timeout = jiffies + HZ/2; /\* timeout in 0.5s \*/

/\* do some work ... \*/

/\* then see whether we took too long \*/

if (timeout > jiffies) {

/\* we did not time out, good ... \*/

} else {

/\* we timed out, error ... \*/

}

Multiple potential overflow issues are here, but let's study one of them: Consider what happens if jiffies wrapped back to zero after setting timeout. Then the first conditional would fail because the jiffies value would be smaller than timeout despite logically being larger. Conceptually, the jiffies value should be a very large numberlarger than timeout. Because it overflowed its maximum value, however, it is now a very small valueperhaps only a handful of ticks over zero. Because of the wraparound, the results of the if statement are switched. Whoops!

Thankfully, the kernel provides four macros for comparing tick counts that correctly handle wraparound in the tick count. They are in <linux/jiffies.h>:

#define time\_after(unknown, known) ((long)(known) - (long)(unknown) < 0)

#define time\_before(unknown, known) ((long)(unknown) - (long)(known) < 0)

#define time\_after\_eq(unknown, known) ((long)(unknown) - (long)(known) >= 0)

#define time\_before\_eq(unknown, known) ((long)(known) - (long)(unknown) >= 0)

The unknown parameter is typically jiffies and the known parameter is the value against which you want to compare.

The time\_after(unknown, known) macro returns true if time unknown is after time known; otherwise, it returns false. The time\_before(unknown, known) macro returns true if time unknown is before time known; otherwise, it returns false. The final two macros perform identically to the first two, except they also return true if the parameters are equal.

The timer-wraparound-safe version of the previous example would look like this:

unsigned long timeout = jiffies + HZ/2; /\* timeout in 0.5s \*/

/\* ... \*/

if (time\_before(jiffies, timeout)) {

/\* we did not time out, good ... \*/

} else {

/\* we timed out, error ... \*/

}

User-Space and HZ

In kernels earlier than 2.6, changing the value of HZ resulted in user-space anomalies. This happened because values were exported to user-space in units of ticks-per-second. As these interfaces became permanent, applications grew to rely on a specific value of HZ. Consequently, changing HZ would scale various exported values by some constantwithout user-space knowing! Uptime would read 20 hours when it was in fact two!

To prevent such problems, the kernel needs to scale all exported jiffies values. It does this by defining USER\_HZ, which is the HZ value that user-space expects. On x86, because HZ was historically 100, USER\_HZ is 100. The macro jiffies\_to\_clock\_t() is then used to scale a tick count in terms of HZ to a tick count in terms of USER\_HZ. The macro used depends on whether USER\_HZ and HZ are integer multiples of themselves. If so, the macro is rather simple:

#define jiffies\_to\_clock\_t(x) ((x) / (HZ / USER\_HZ))

A more complicated algorithm is used if the values are not integer multiples.

Finally, the function jiffies\_64\_to\_clock\_t() is provided to convert a 64-bit jiffies value from HZ to USER\_HZ units.

These functions are used anywhere a value in ticks-per-seconds needs to be exported to user-space. Example:

unsigned long start;

unsigned long total\_time;

start = jiffies;

/\* do some work ... \*/

total\_time = jiffies - start;

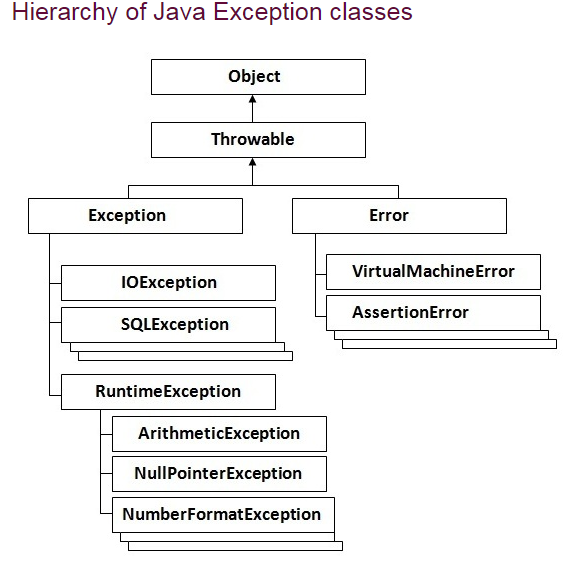
printk("That took %lu ticks\n", jiffies\_to\_clock\_t(total\_time));

##############################################################################

**JAVA**

**Exception Handling in Java:**

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.



**Checked and Unchecked exceptions:**

1. **Checked Exception :** Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using [**try-catch block**](http://beginnersbook.com/2013/04/try-catch-in-java/) or it should declare the exception using [**throws keyword**](http://beginnersbook.com/2013/04/difference-between-throw-and-throws-in-java/), otherwise the program will give a compilation error. It is named as ***checked exception*** because these exceptions are ***checked*** at Compile time.

Few Checked Exceptions –

SQLException

IOException

FileNotFoundException

1. **Unchecked Exception :** Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn’t handle/declare that exception, the program won’t give a compilation error. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of **RuntimeException** class.

Few UncheckedExceptions –

NullPointerException

ArrayIndexOutOfBound

IllegalArgumentException

IllegalStateException

1. **Error :**

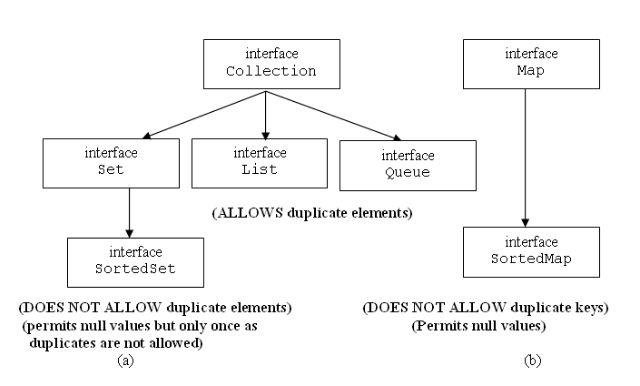
Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

**NOTE:**

If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

[**COLLECTION FRAMEWORK IN JAVA**](http://way2java.com/category/collections/)**,**[**GENERAL**](http://way2java.com/category/collections/general-collections/)

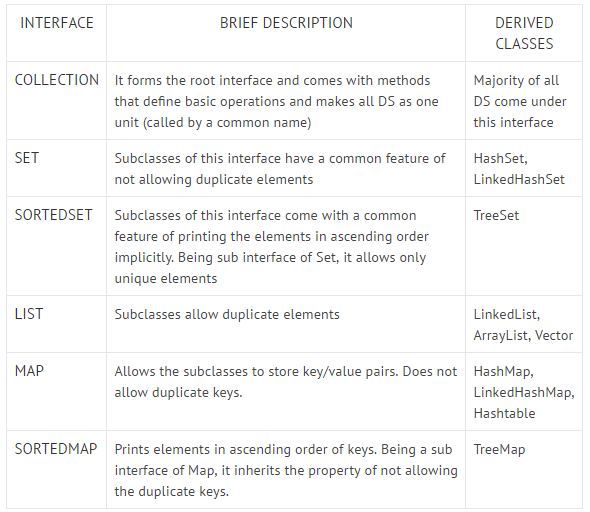
Following hierarchy gives all the interfaces of Collections framework.



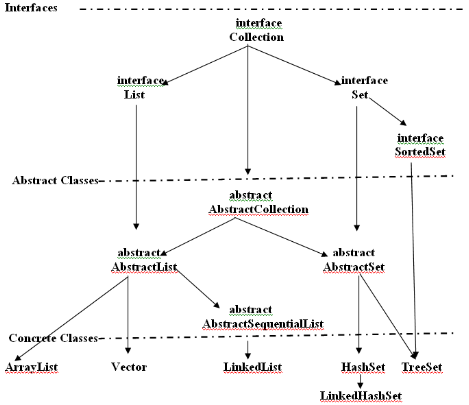
**NOTE :**

1. **Map** and **SortedMap** form a separate hierarchy and not connected with[Collection](http://way2java.com/collections/java-interface-collection/) interface.
2. Framework name is collections and the interface name is [Collection](http://way2java.com/collections/java-interface-collection/). Do not get confused; later, we get one more class called [Collections](http://way2java.com/collections/collections-api-methods/).

**Description of fundamental interfaces of Collections Hierarchy**



**Following figure displays the main interfaces with relation to their derived classes.**

****

**ArrayList :** It supports dynamic arrays that can grow as needed. In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink. An ArrayList can **dynamically increase or decrease in size**. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

ArrayList has the **constructors** shown here:

ArrayList( )

ArrayList(Collection<? extends E> c)

ArrayList(int capacity)

void ensureCapacity(int *cap*) : you can increase the capacity of an ArrayList object manually by calling ensureCapacity( ).

void trimToSize( ) : to reduce the size of the array

Obtaining an Array from an ArrayList

Object[ ] toArray( )

<T> T[ ] toArray(T *array*[ ]) : Returns an array of elements that have the same type as T.

**Example**

ArrayList<Integer> al = **new** ArrayList<Integer>();

// Get the array.

Integer ia[] = **new** Integer[al.size()];

ia = al.toArray(ia);

**LinkList:** The LinkedList class extends AbstractSequentialList and implements the List, Deque, and Queue interfaces. It provides a linked-list data structure. LinkedList has the two constructors shown here:

LinkedList( )

LinkedList(Collection<? extends E> c)

Because **LinkedList** implements the **Deque** interface, you have access to the methods defined by Deque. For example, to add elements to the start of a list you can use **addFirst**( ) or **offerFirst**( ). To add elements to the end of the list, use

**addLast**( ) or **offerLast**( ). To obtain the first element, you can use **getFirst**( ) or **peekFirst**( ). To obtain the last element, use **getLast**( ) or **peekLast**( ). To remove the first element, use **removeFirst**( ) or **pollFirst**( ).

To remove the last element, use **removeLast**( ) or **pollLast**( ).

To insert items at a specific location, use the **add(int, E).**

**The HashSet Class**

It creates a collection that uses a hash table for storage. The advantage of hashing is that it allows the execution time of

add( ), contains( ), remove( ), and size( ) to remain constant even for large sets.

The following **constructors** are defined:

HashSet( )

HashSet(Collection<? extends E> c)

HashSet(int capacity)

HashSet(int capacity, float fillRatio)

The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward.

It is important to note that HashSet does not guarantee the order of its elements.

**The LinkedHashSet Class**

Its constructors parallel those in HashSet. LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted. This allows **insertion-order iteration** over the set.

**The TreeSet Class**

**TreeSet** extends **AbstractSet** and implements the **NavigableSet** interface. It creates a collection that uses a **tree** for storage. Objects are stored in sorted, **ascending order**. Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

**TreeSet has the following constructors:**

TreeSet( )

TreeSet(Collection<? extends E> c)

TreeSet(Comparator<? super E> comp)

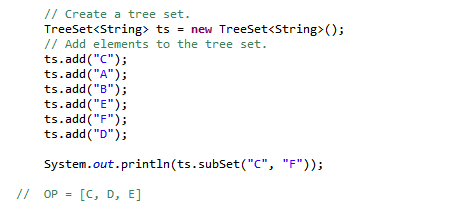
TreeSet(SortedSet<E> ss)

The third form constructs an empty tree set that will be sorted according to the **comparator**

specified by comp.

**subSet( )** is used to obtain a subset of treeSet that contains the elements between C (**inclusive**) and F (**exclusive**).

**Example:**

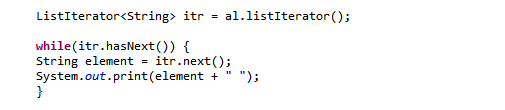


**Accessing a Collection via an Iterator**

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements. Iterator and ListIterator are generic interfaces which are declared as shown here:

interface Iterator<E>

interface ListIterator<E>

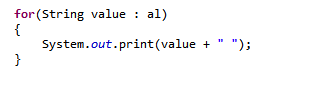


**The For-Each Alternative to Iterators**

If you won’t be modifying the contents of a collection or obtaining elements in reverse

order, then the for-each version of the for loop is often a more convenient alternative to

cycling through a collection than is using an iterator.



**Working with Maps**

A map is an object that stores associations between keys and values, or **key/value pairs**. Given

a key, you can find its value. Both keys and values are objects. The keys must be unique,

but the values may be duplicated. Some maps can accept a null key and null values, others

cannot.

interface Map<K, V>

Maps revolve around two basic operations: get( ) and put( ).

As mentioned earlier, although part of the Collections Framework, maps are not,

themselves, collections because they do not implement the Collection interface. However, you

can obtain a collection-view of a map. To do this, you can use the **entrySet**( ) method.

It returns a Set that contains the elements in the map. To obtain a collection-view of the keys,

use **keySet**( ). To get a collection-view of the values, use **values**( ).

**The SortedMap Interface**

The SortedMap interface extends Map. It ensures that the entries are maintained in

ascending order based on the keys. SortedMap is generic and is declared as shown here:

interface SortedMap<K, V>

Sorted maps allow very efficient manipulations of **submaps** (in other words, subsets of a

map). To obtain a **submap**, use **headMap**( ), **tailMap**( ), or **subMap**( ). The submap

returned by these methods is backed by the invoking map. Changing one changes the

other. To get the first key in the set, call **firstKey**( ). To get the last key, use **lastKey**( ).

**The NavigableMap Interface**

The NavigableMap interface extends SortedMap and declares the behavior of a map

that supports the retrieval of entries based on the closest match to a given key or keys.

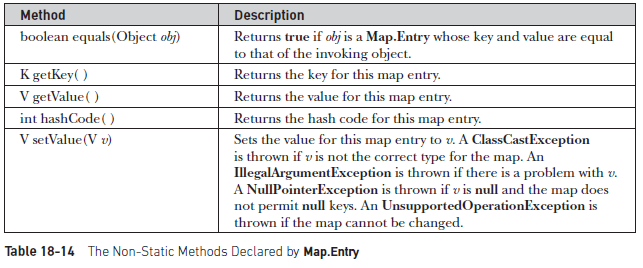
**The Map.Entry Interface**

The **Map.Entry** interface enables you to work with a map entry. Recall that the **entrySet**( )

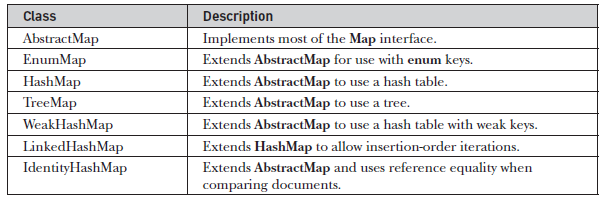
method declared by the Map interface returns a Set containing the map entries. Each of

these set elements is a Map.Entry object. Map.Entry is generic and is declared like this:

interface Map.Entry<K, V>



**The Map Classes**

****

Notice that AbstractMap is a superclass for all concrete map implementations.

WeakHashMap implements a map that uses “**weak keys**,” which allows an element in a

map to be **garbage-collected** when its key is otherwise **unused.**

**The HashMap Class**

It uses a hash table to store the map. This allows the execution time of get( ) and put( ) to remain constant even for large sets. HashMap is a generic class that has this declaration:

**Note:** A hash map does not guarantee the order of its elements. Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

The following constructors are defined:

HashMap( )

HashMap(Map<? extends K, ? extends V> m)

HashMap(int capacity)

HashMap(int capacity, float fillRatio)

**Example**

// Create a hash map.

HashMap<String, Double> hm = **new** HashMap<String, Double>();

// Put elements to the map

hm.put("John Doe", **new** Double(3434.34));

hm.put("Tom Smith", **new** Double(123.22));

hm.put("Jane Baker", **new** Double(1378.00));

hm.put("Tod Hall", **new** Double(99.22));

hm.put("Ralph Smith", **new** Double(-19.08));

// Get a set of the entries.

Set<Map.Entry<String, Double>> set = hm.entrySet();

// Display the set.

**for**(Map.Entry<String, Double> me : set) {

System.*out*.print(me.getKey() + ": ");

System.*out*.println(me.getValue());

**The TreeMap Class**

The TreeMap class extends AbstractMap and implements the NavigableMap interface. It creates maps stored in a **tree** **structure**. A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in **ascending** **key** **order**.

The following TreeMap constructors are defined:

TreeMap( )

TreeMap(Comparator<? super K> comp)

TreeMap(Map<? extends K, ? extends V> m)

TreeMap(SortedMap<K, ? extends V> sm)

The **second** form constructs an empty tree-based map that will be sorted by using the **Comparator** comp.

**The LinkedHashMap Class**

LinkedHashMap extends HashMap. It maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map.

LinkedHashMap defines the following **constructors**:

LinkedHashMap( )

LinkedHashMap(Map<? extends K, ? extends V> m)

LinkedHashMap(int capacity)

LinkedHashMap(int capacity, float fillRatio)

LinkedHashMap(int capacity, float fillRatio, boolean Order)

The last form allows you to specify whether the elements will be stored in the linked list **by insertion order**, or by **order of last access**. If Order is true, then access order is used. If Order is false, then insertion order is used.

**The IdentityHashMap Class**

It is similar to HashMap except that it uses reference equality when comparing elements.

**The EnumMap Class**

EnumMap extends AbstractMap and implements Map. It is specifically for use with keys of an enum type. It is a generic class that has this declaration:

class EnumMap<K extends Enum<K>, V>

EnumMap defines the following **constructors**:

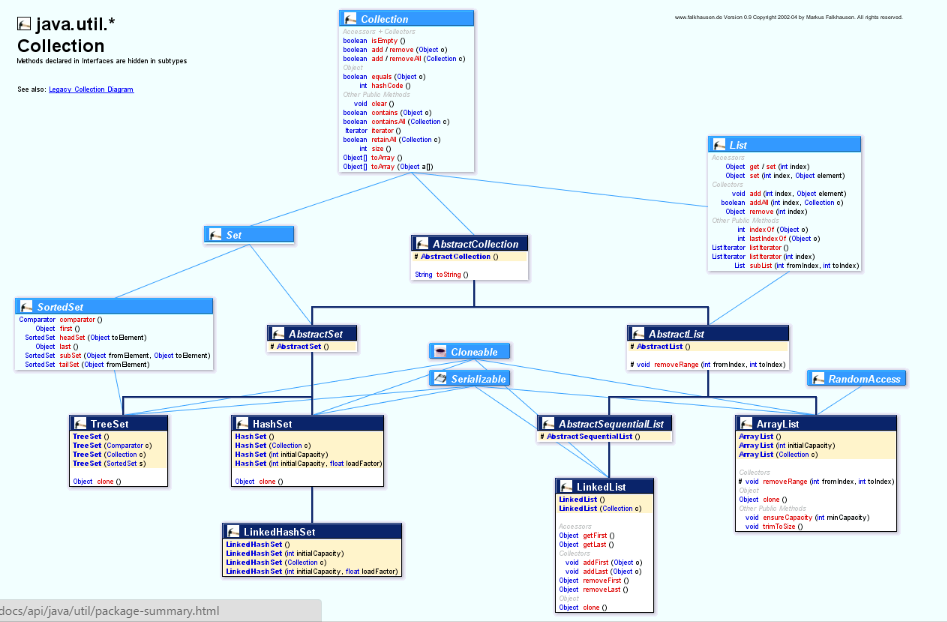
EnumMap(Class<K> kType)

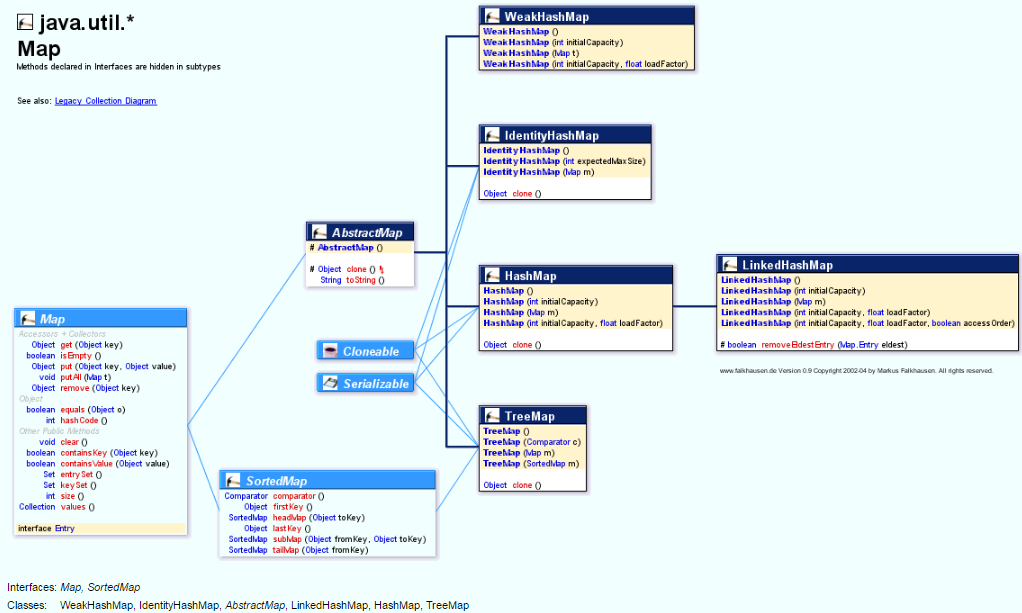
EnumMap(Map<K, ? extends V> m)

EnumMap(EnumMap<K, ? extends V> em)

The first constructor creates an empty EnumMap of type kType. The second creates an EnumMap map that contains the same entries as m. The third creates an EnumMap initialized with the values in em. EnumMap defines no methods of its own.

**Detailed Hierarchy**

****

****

**Comparators**

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what “sorted order” means. By default, these classes store their elements by using what Java refers to as “**natural ordering**,” which is usually the ordering that you would expect (A before B, 1 before 2, and so forth). If you want to order elements a different way, then **specify a Comparator** when you construct the set or map.

Comparator is a generic interface that has this declaration: Here, **T** specifies the type of objects being compared.

**interface Comparator<T>**

The Comparator interface defined only two methods: **compare**( ) and **equals**( ).

The compare( ) method, shown here, compares two elements for order:

**int compare(T obj1, T obj2)**

obj1 and obj2 are the objects to be compared. Normally, this method returns **zero** if the objects are equal. It returns a **positive** value if obj1 is greater than obj2. Otherwise, a **negative** value is returned. The method can throw a ClassCastException if the types of the objects are not compatible for comparison.

By implementing compare( ), you can alter the way that objects are ordered.

The **equals**( ) method, shown here, tests whether an object equals the invoking comparator:

**boolean equals(object obj)**

Here, obj is the object to be tested for equality. The method returns **true** if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns **false**.

**Example:**// A reverse comparator for strings.

**class** MyComp **implements** Comparator<String> {

**public** **int** compare(String aStr, String bStr) {

// Reverse the comparison.

**return** bStr.compareTo(aStr);

}

// No need to override equals or the default methods.

}

**class** CompDemo {

**public** **static** **void** main(String args[]) {

// Create a tree set.

TreeSet<String> ts = **new** TreeSet<String>(**new** MyComp());

// Add elements to the tree set.

ts.add("C");

ts.add("A");

ts.add("B");

ts.add("E");

ts.add("F");

ts.add("D");

// Display the elements.

**for** (String element : ts)

System.*out*.print(element + " ");

System.*out*.println();

}

}

// OP : F E D C B A

**Example:** Using comparator to compare treeMap.

This Example user entrySet() to get a set from TreeMap and compare method is overridden to compart values. Note that comparison is always done on the basis of keys which are of Double type in our case.

######################################################################################

**GENERICS**

**Why Generics?**

Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called **generic**, as in **generic class** or **generic method**.

It is important to understand that **Java has always given** **you the ability to create generalized** classes, interfaces, and methods by operating through references of type Object. Because Object is the superclass of all other classes, an Object reference can refer to any type object. Thus, in pre-generics code, generalized classes, interfaces, and methods used Object references to operate on various types of objects. The **problem** was that they could not do so with **type** **safety**. Generics added the type safety that was lacking. They also streamlined the process, because it is no longer necessary to explicitly employ **casts** to translate between Object and the type of data that is actually being operated upon. With generics, all **casts are automatic** and implicit.

**Generic Class**

A generic class is defined with the following format:

class name<T1, T2, ..., Tn> { /\* ... \*/ }

**EXAMPLE:**



**Raw Types**

A raw type is the name of a generic class or interface without any type arguments. For example, given the generic Box class:

public class Box<T> {

public void set(T t) { /\* ... \*/ }

// ...}

If the actual type argument is omitted, you create a raw type of Box<T>:

Box rawBox = new Box();

Therefore, Box is the raw type of the generic type Box<T> For backward compatibility, assigning a parameterized type to its raw type is allowed:

Box<String> stringBox = new Box<>();

Box rawBox = stringBox; // OK

**Generic Methods**

*Generic methods* are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared.

Example:



**Bounded Type Parameters**

When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived. This is accomplished through the use of an extends clause when specifying the type parameter, as shown here:

**<T extends superclass>**

****

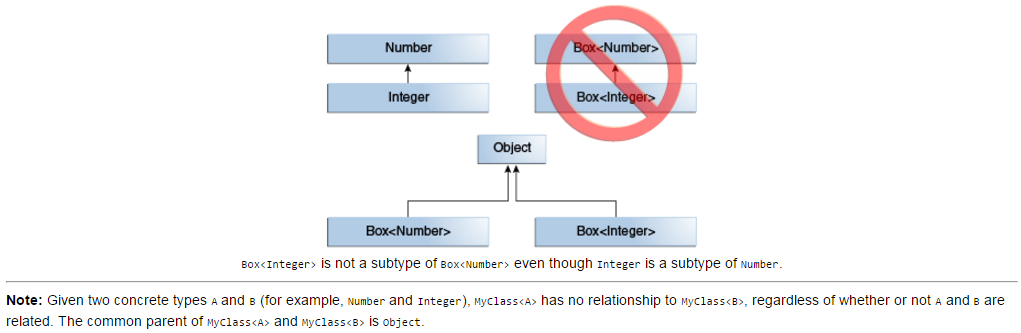
**Generics, Inheritance, and Subtypes**

Now consider the following method:

public void boxTest(Box<Number> n) { /\* ... \*/ }

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is Box<Number>. But what does that mean? Are you allowed to pass in Box<Integer> or Box<Double>, as you might expect? The answer is "no", because Box<Integer> and Box<Double> are not subtypes of Box<Number>.

This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn.



**Wildcards**

In generic code, the question mark (?), called the wildcard, represents an unknown type**.**

**Upper Bounded Wildcards**

For example, say you want to write a method that works on List<Integer>, List<Double>, andList<Number>; you can achieve this by using an upper bounded wildcard.

To write the method that works on lists of Number and the subtypes of Number, such as Integer, Double, and Float, you would specify **List<? extends Number>.** The termList<Number> is **more** **restrictive** than List<? extends Number> because the former matches a list of type Number only, whereas the latter matches a list of type Number or any of its **subclasses.**

**Unbounded Wildcards**

Consider the following method, printList:



The goal of printList is to print a list of any type, but it fails to achieve that goal — it prints only a list of Object instances; it cannot print List<Integer>, List<String>,List<Double>, and so on, because they are not subtypes of List<Object>. To write a generic printList method, use List<?>:

****

**Lower Bounded Wildcards**

To write the method that works on lists of Integer and the supertypes of Integer, such as Integer, Number, and Object, you would specify List<? super Integer>. The termList<Integer> is more restrictive than List<? super Integer> because the former matches a list of type Integer only, whereas the latter matches a list of any type that is a supertype of Integer.

