**Generics**

The main objective of generics are to provide Type Safety and to resolve type Casting problems.

**Case1 Type Safety**

Arrays are type safe that is we can give the guarantee to type of element present in the array. For ex. If our programing requirement is to hold only String type of object we can choose String Array. By mistake if we are trying to add any other type of objects we will get compile time error.

Collections are not type safe. That is we can’t give the guarantee for type of element present inside collection. For example if our programming requirement is to hold only String type object if we choose ArrayList by mistake if you are trying to add other type of objects. We won’t get any compile time error. The program may fail at runtime when storing the object to required type.

**Case2 Type Casting**

In the case of Arrays at the time of retrieval it is not required to perform type casting because there is a guarantee for the type of elements present inside array.

In the case of collection at time of retrieval compulsory. we should perform type casting because there is no guarantee for the type of element present inside collection. Hence typecasting is bigger headache in collection,

**To overcome above problem in collections, Sun people introduced generics concept in 1.5V.**

**Conclusion**

Polymorphism concept only applicable for base type but not for parameter Type. Usage of parent reference to hold child object is the concept of polymorphism.

ArrayList<Object> al = new arrayList<String>(); //**incompatible type found exception**

For the type parameter we can provide any class or interface name but no primitives, if we are trying to provide primitive then we will get compile time error.

ArrayList<int> al = new ArrayList<int>(); //**unexpected type found exception**

**Bounded Type**

We can Bound Type parameter for a particular range using extends keyword such type are called Bounded Types.

Ex

Class Test<T extends Number> {  
 T a , b;

} // we can only pass either number or its child classes only. We can define bounded type only by extends keyword even if the Type is interface.

We can define bounded types even in combination also.

Class Test < T extends number & Runnable){

// as a type parameter we can take anything which should be child class of number and implements Runnable Interface.

}

Class Test <T extends Runnable & Number> //INVALID  
Class Test<T extends Number and Thread> //INVALID because we can’t extend more than one class simultaneously.

**Note**

As a type parameter <T> we can take any valid java identifier but it is convection to use T.  
We can declare any number of Types as a parameter. Class<A,B,C>

**Generic Method and Wildcard characters**

m1(ArrayList<T> p);

m1(ArrayList<?> p); we can call this method by passing ArrayList of any Type. But within a method we cant add anything to the list except null. Because we don’t know the type exactly. Null is allowed it is valid value for any type. This type of methods are best suitable for read only operation.

m1(ArrayList<? Extends X> p); X can be either class or interface. If X is a class then we can call this method by passing ArrayList of either X type or its child classes. IF X is an interface then we can call this method by passing either ArrayList if X type or its implementation classes.   
 But within method we can’t add anything to the list except null. . Because we don’t know the type exactly. This type of methods are best suitable for read only operation.

m1(ArrayList<?> Super X p); X can be either class or interface. If X is a class then we can call this method by passing ArrayList of either X type or its super Classes. If X is an interface then we can call this method by passing ArrayList of either X type or super class of implementation class of X. But within the method we can add X type of objects and null to the list.

ArrayList<String> l = new ArrayList<String>();  
 ArrayList<?> l = new ArrayList<String>();  
 ArrayList<? extends Number> l = new ArrayList<Integer>();  
 ArrayList<? extends Number> l = new ArrayList<String>();  
 ArrayList<? Super String> l = new ArrayList<Object>();  
 ArrayList<? > l = new ArrayList<?>(); // required class or interface without bounds.  
 ArrayList<? > l = new ArrayList<? extends Number>();

Generic Method

We can declare Type parameter either at class level or at method level. We have to declare Type parameter just before return Type. We can define bounded Type at method level also.

Class Test {

Public <T extends Number & Runnable>void m1(T p){  
We can use T anywhere within this method based on our requirement

}  
}

**Note**

Generics concept only applicable at Compile Time… after compiling all Generics will be removed

ArrayList l = new ArrayList<T>();  
 ArrayList l = new ArrayList();

Above two constructors are Equal at RunTime.

ArrayList<T> l = new ArrayList<T>();  
 ArrayList<T> l = new ArrayList();

Above two constructors are Equal at RunTime. For these objects we can add only String type of objects.

**Thread**

One process can spawn multiple Thread and share same memory in Java. Each thread has its own stack. Thread is separate flow of execution

**Multi-Tasking**

Executing several task simultaneously is the concept of multi – Tasking. There are two type:

* Process-based multitasking.
* Thread- based multitasking.

**Process based Multitasking**

Executing several task simultaneously where each task is separate independent program (process) is called process based multitasking.

E.G

While typing java program in an editor we can listen audio songs from same system at the same time we can download a file from net. All these tasks will be executed simultaneously and independent of each other hence it is process based multi-tasking

Process based multi-tasking is best suitable at OS level.

**Thread based multitasking**

Executing several tasks simultaneously where each task is separate independent part of same program is called thread based multi-tasking.

Thread based multi-tasking is best suitable at programmatic level.

Whether it Process-based or thread-based, The main objective of Multi-Tasking is to reduce response time of the system and to improve performance.

**Multi-Threading**

The main important application areas of multi-threading are:

1. To develop multimedia graphics
2. To develop animation, video games.
3. To develop webserver and application server etc.

Note

When compared to old languages developing multi-threaded application in java is very easy because Java provides inbuilt support for multi-threading with rich api. [Thread , Runnable, Thread Group…]

**Defining thread**

We can define thread by following two ways

* By extending Thread class
* By implementing Runnable Interface

**Case 1**

**Thread scheduler**

It is the part of JVM. It is responsible to schedule thread. That is if multiple threads are waiting to get chance of execution then in which orrder threads will executed is decided by thread scheduler. We cant except exact algorithm from thread scheduler it is varied from JVM to JVM hence we cant except thread execution order and exact output. Hence whenever situation comes to multi-threading there is no guarantee for exact output. But we can provide several possible outputs.

**Case 2**

**Difference between t.start() and t.run()**

In case of **t.start()** new thread will be created which is responsible for the execution of run method. But in case of **t.run()** a new thread won’t be created and run method will be executed just like a normal method call by main thread.

**Case 3**

**Importance of t.start() method**

Thread class start method is responsible to register thread with Thread scheduler and all other mandatory activities. Hence without executing thread class **start()** method there is no chance of starting a new thread in java. Due to this, Thread class **start()** method is considered as heart of multi-threading.

**Case 4**

**Overloading of run method**

Overloading of run method is always possible but thread class **start method** can invoke no argument run method. The overloaded method we have to call explicitly like a normal method call.

**Case 5**

**Overriding of run method and start method.**

If we are not overriding run method thread class run method will be executed which has empty implementation. If we are overriding start method then it will be just normal method. It is not recommended to over-ride start method then go for multi-threading concept.

Case 6

After starting a thread if we are trying to restart the same thread then we will get run-time exception saying **IllegaTthreadStateException.**

**Note**

Main difference is that when program calls start() method a **new Thread**is created and code inside run() method is executed in new Thread while if you call run() method directly **no new Thread is created** and code inside run() will execute on **current Thread**.

**Define a thread by implementing Runnable Interface**

Runnable interface present in java.lang package and it contains only **run() method**.

MyRunnable myr = new MyRunnable();  
 Thread t = new Thread(myr);  
 t.start();

**Case study**

MyRunnable myr = new MyRunnable();  
 Thread t = new Thread(myr);   
 Thread t2 = new Thread();

Case 1:

t2.start() is will have an empty implementation as thread class is not linked to MyRunnable class.

Case 2:

Myr.start(); compiletime error cause Runnable interface as only one method which is run method();

**Which approach is best to define a thread?**

Among two ways of defining thread implements Runnable approach is recommended. Because in first approach our class always extends thread class, there is no chance of extending any other class hence we are missing inheritance benefit. But in the second approach while implementing runnable interface we can extend any other class hence we won’t miss any inheritance benefit.

**Valid approach**

MyThread t = new MyThread();  
 Thread t1 = new Thread(t);  
 t1.start();

Its valid because MyThread class extends thread class and that thread class implements runnable class. So technically Mythread is an runnable object.

**Thread class constructors:**

Thread t = new Thread();  
 Thread t = new Thread(Runnable r);  
 Thread t = new Thread(String name);  
 Thread t = new Thread(Runnable r. String name);  
 Thread t = new Thread(ThreadGroup g. String name);  
 Thread t = new Thread(ThreadGroup g, Runnable r.);  
 Thread t = new Thread(ThreadGroup g, Runnable r. String name);  
 Thread t = new Thread(ThreadGroup g, Runnable r. String name, long stackSize);

**Getting and Setting name of thread**

Every thread in java has some name. it may be default name generated by JVM. Or customized name provided by programmer.

We can get and set a name of thread by using following two methods of thread class.

Public final string getName();  
 public final void setName(String name);

Note

We can get current using thread name by using **Thread.currentThread().getName();**

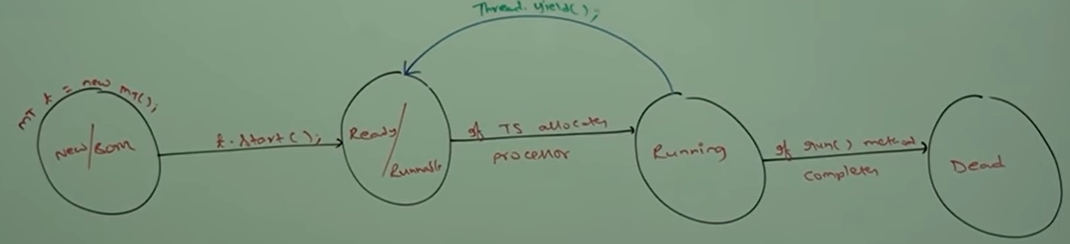
**Java Concurrency – yield(), sleep() and join() methods**

**Yield() –**

Yield method causes to pause current executing thread to give chance for waiting thread of same priority. If there is no waiting thread or all waiting thread have low priority then same thread can continue its execution. If multiple thread waiting with same priority then which waiting thread will get the chance we can’t except it depends on thread scheduler.

The thread which is yielded, when it will get the chance once again is depends on thread scheduler and we can’t except order exactly.

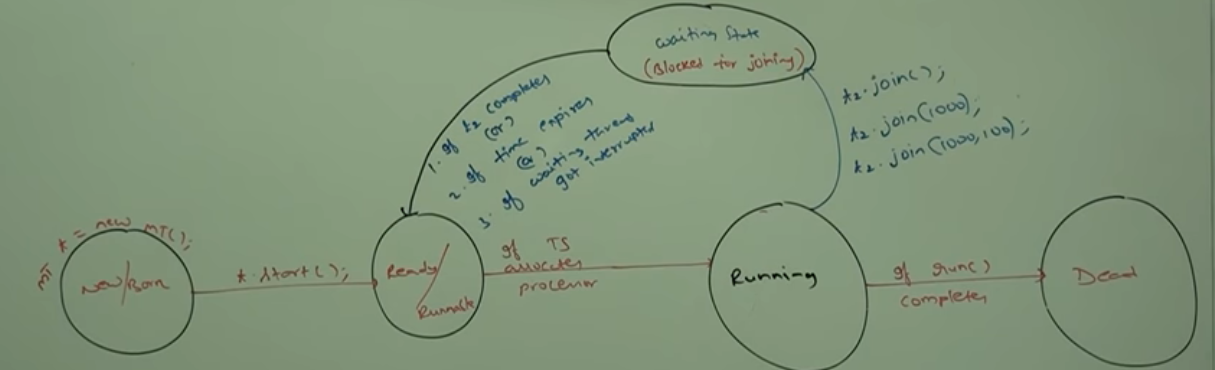
**Public static native void yield();**

****

**join():**

if thread wants to wait until completing some other thread then we should go for join method. If a thread t1 wants to wait until completing t2 then t1 has to call **t2.join().** If t1 executes t2.join then immediately t1 will be entered into waiting state until t2 completes. Once t2 completes then t1 will continue is execution.

**Public final void join(); throws IntereptedException  
Public final void join(long milisec); throws IntereptedException  
Public final void join(long milisec, int nanosec); throws IntereptedException**

****

**Note**

Main thread can assign to variable by using **Thread** as reference.

**Thread mt = Thread.currentThread**.

If main thread call join method on child thread object and child thread call join method on main thread then both thread will wait forever and program will be struck. This is something like deadlock.

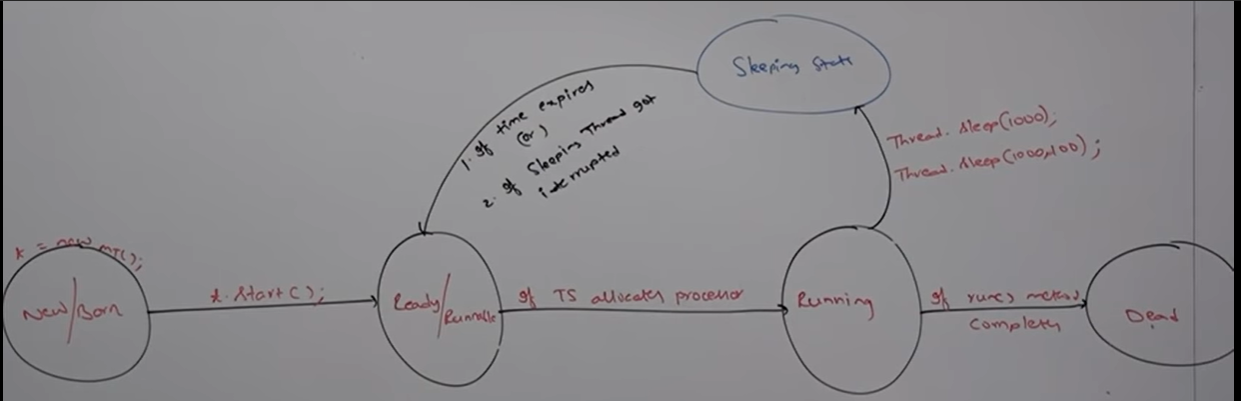
If thread calls join method on same thread itself then the program will be struck. In this case thread as to wait infinite amount of time. **Thread.currentThread().join();**

**Sleep():**

If a thread doesn’t want to perform any operation for particular amount of time then we should go for sleep method. It is useful while do slideshow.

Public static native void sleep(long ms); **throws IntereptedException**

Public static void sleep( long ms, int ns); **throws IntereptedException**

****

**How a thread can interrupt another thread?**

**interrupt() method : If any thread is in sleeping or waiting state then using interrupt() method, we can interrupt the execution of that thread by showing InterruptedException. A thread which is in the sleeping or waiting state can be interrupted with the help of interrupt() method of Thread class.**

**Note**

**Whenever we are calling interrupt method, if the target is not in the sleeping state or waiting state. Then there is no effect of interrupt call immediately. Interrupt call will waited until target thread entered into sleeping or waiting state. If the target enters into sleeping or waiting state then immediately interrupt call will interrupt the target thread.**

**If the target thread never entered into sleeping or waiting state in its life time then there is no impact of interrupt call. This is the only case where interrupt call will be wasted.**

**Thread Synchronization**

**Synchronized is a modifier applicable only for method and blocks. If a multiple threads are trying to operate simultaneously on the same java object then there may be a chance of data inconsistency problem. To overcome this problem we should go for synchronized keyword. If a method or block declared as synchronized then at a time only one thread is allowed to execute method or block on given object. data inconsistency problem will be resolved.**

**The main advantage of synchronized keyword is we can resolve data inconsistency problem. The main disadvantage of synchronized keyword it increases waiting time for thread and creates performance problem. So synchronized keyword is not recommended to use unless requirement require.**

**Internally synchronized concept is implemented using lock. Every object in java has a unique lock. Whenever we are using synchronized keyword then only lock concept will come into picture.**

* **If a thread wants to execute a synchronized method on the given object first it has to get lock of that object.**
* **Once thread got lock then It allowed to execute any synchronized method on that object.**
* **Once method execution completes automatically thread releases lock.**
* **Acquiring and releasing lock internally takes care by JVM and programmer not responsible for this activity.**

**While a thread executing synchronized method on the given object. The remaining thread are not allowed to execute any synchronized method simultaneously on the same object. But remaining thread are allowed to execute non synchronized method simultaneously.**

**Lock concept is implemented based on object but not based on method.**

**Case study**

**Display d1 = new Display();  
 Display d2 = new Display();  
 MyThread t1 = new MyThread(d1,”ABD”);  
 MyThread t2 = new MyThread(d2,”Kohile”);  
 t1.start(); //d1.wish(ABD);  
 t2.start(); //d2.wish(Kohile);**

**Even through wish() method is synchronised we will get irregular output because threads are operating on different java objects.**

**Conclusion**

**If multiple threads are operating on same java object then synchronization is required. If multiple threads are operating on multiple java object then synchronization is not required.**

**Class level Lock**

**Every class in java has unique lock which is nothing but class level lock. If thread wants execute a static synchronized method then thread require class level lock. Once thread got class level lock then it is allowed to execute any static synchronized method of that class. Once method execution completes automatically thread releases the lock.**

**Synchronized Block**

**If very few lines of the code required synchronization then it is not recommended to declare entire method as synchronized. We have to enclose those few lines of code by using synchronized block.**

**The main advantage of synchronized block over synchronized method is it reduces waiting time of thread and improves performance of application.**

**We can declare synchronized block as follows**

|  |  |  |
| --- | --- | --- |
| **To get lock of current object** | **To get lock of particular object “obj”;** | **To get class level lock** |
| Synchronized(this){  ----  ------ }  If a thread got lock of current object then only it is allowed to execute this area. | Synchronized(obj){  ----  ------ }  If a thread got lock of particular object ‘obj’ then only it is allowed to execute this area | Synchronized(Display.class){  ----  ------ }  If a thread got class level lock of **Display** class, then only it is allowed to execute this Area. |

**Note**

If multiple objects are operating simultaneously on same object then there may be chance of data consistency problem called **race condition**. We can overcome this situation by using **synchronized keyword.**

**Is thread can acquire multiple locks or not?**

Yes. From different objects

Class x {  
 public synchronized void m1() { // here thread has lock of X object   
 Y y = new Y();  
 synchronized(y){ // here thread has lock of x and y  
 Z z = new Z();  
 synchronized(z){ // here thread has lock of X,Y,Z  
 ` }  
 }  
 }

**Inter Thread communication**

Two thread can communicate with each other by using **wait(), notify() and notifyALl() methods.** The thread which is expecting update is responsible to call **wait method** then that thread will enter into waiting state. The thread which is responsible to perform update. After performing update it is responsible to call **notify() method**. Then waiting thread will get the notification and continue its execution with those updated items.

**Note**

wait(), notify(), notifyAll() methods present in object class not in thread class because thread can call this methods on any java object.

To call wait(), notify(), notifyAll() methods on any object, thread should be owner of that object that is thread should has lock of that object that is the thread should inside synchronized area. Hence we can call wait(), notify(), notifyAll() methods only form synchronized area otherwise we will get runtime exception saying **IllegalMonitorStateException**

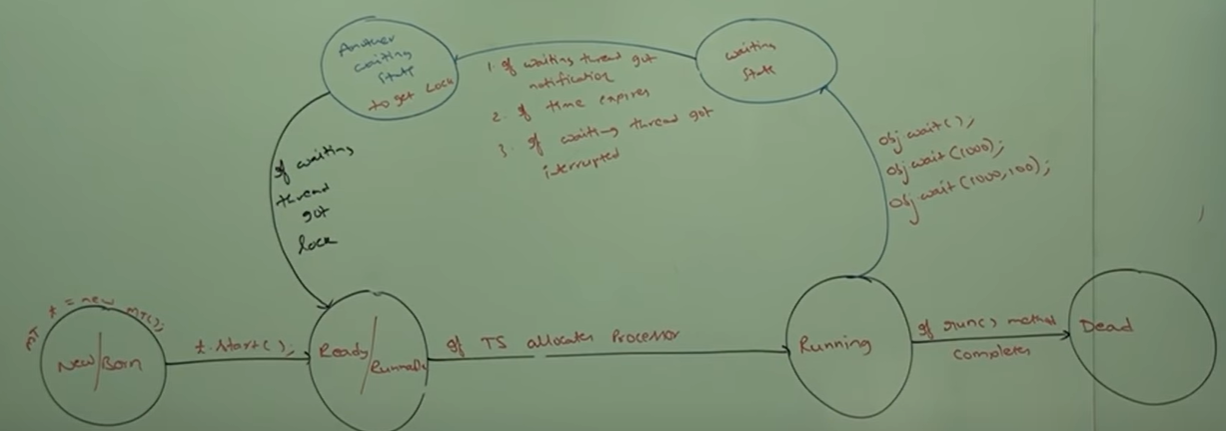
If a thread calls **wait()** **metho**d on any object it immediately releases the lock of particular object and enter into waiting state.

If thread calls **notify() method** on any object it releases the lock of that object may not immediately except wait(), notify(), notifyAll() there is no other method where thread releases lock.

**Methods**

Public final void wait(); throws IE  
public final native void wait(long ms); throws IE  
Public final void wait(long ms, int ns); throws IE

Public final native void notify();  
Public final native void notifyAll();



**Producer and consumer problem**

Producer thread is responsible to produce items to queue. And consumer thread is responsible to consume items from the queue.

If queue is empty consumer thread will call wait method and entered into waiting state.

After producing items to the queue by producer. Producer thread is responsible to call notify method then waiting-consumer will get that notification and continue its execution with updated items.

**Difference between notify and notifyAll()**

We can use **notify()** method to give notification for only one waiting thread. If multiple thread are waiting then only one thread will be notified and the remaining threads have to wait for further notification.   
Which thread will notify we can’t except. It depends on JVM.

We can use **notifyAll()** method to give notification for all waiting threads of a particular object. Even through multiple threads notified and execution will be performed one by one because threads require lock and only lock is available.

**Deadlock**

If two threads are waiting for each other forever, such type of infinite waiting is called deadlock.

Synchronized keyword is the only reason for dead lock situation. By using synchronized keyword we have to take special care. There are no resolution technique for deadlock but several prevention technique are available.

Daemon Thread

The threads which are executing in the background are called Daemon threads.

Example: Garbage collector, Signal dispatcher, attach listener etc.

The main objective of daemon thread is to provide support for non-daemon thread (Main thread). If main thread runs with low memory then JVM will run Garbage Collector to destroy useless object. So that number of byte of free memory will be improved. With this free memory main thread can continue its execution.