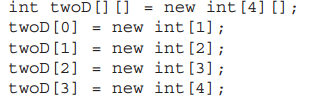
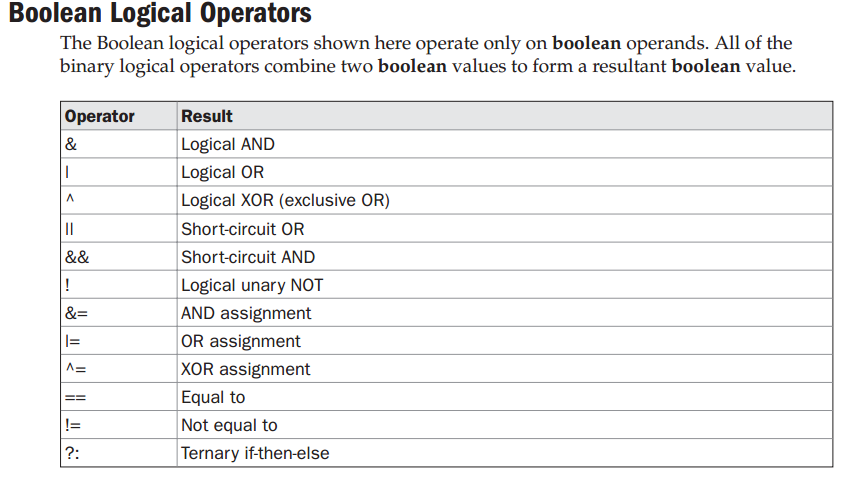
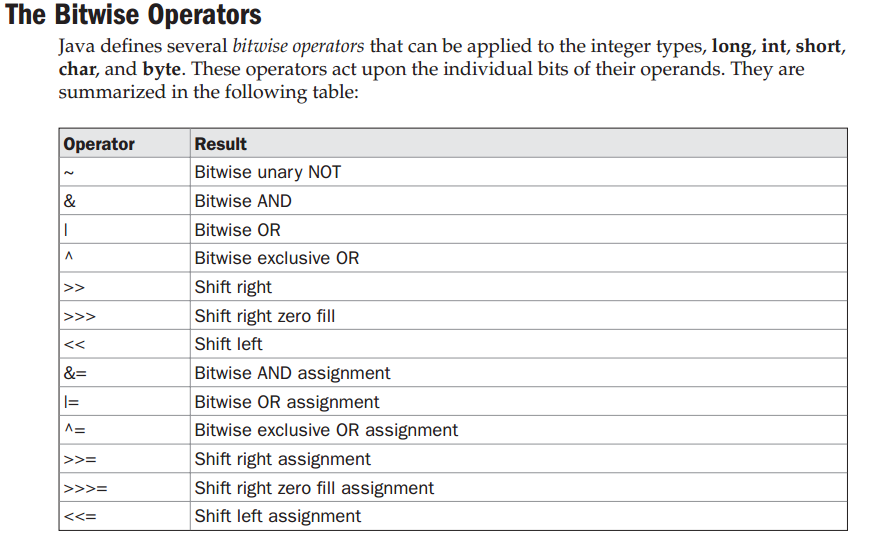
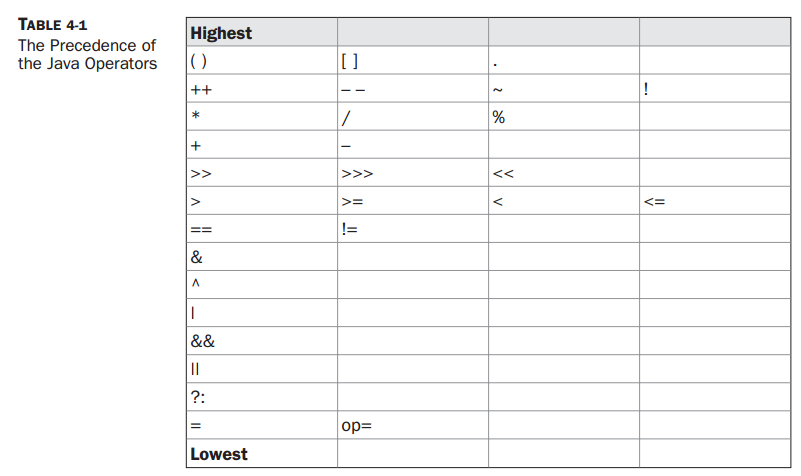


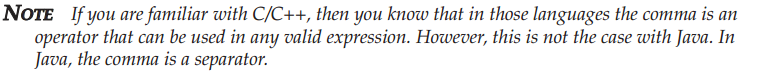
**DECLARING VARIABLE LENGHTS OF SECOND DIMENTION IN 3D ARRAY**





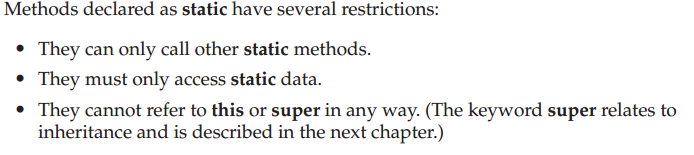


**Foreach:** 

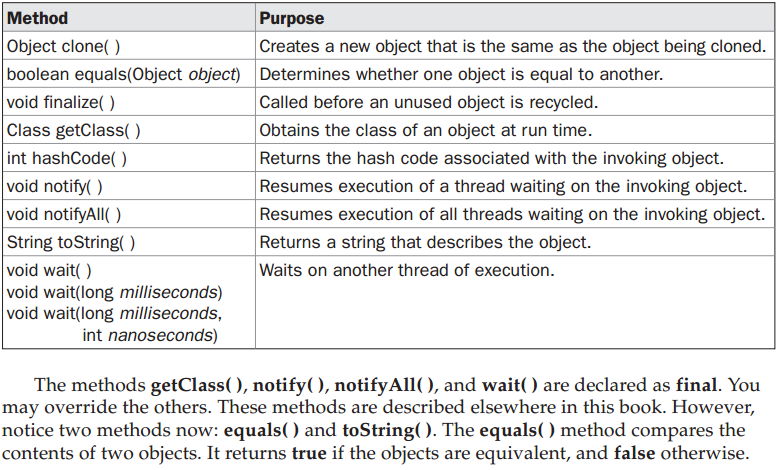


**Jump statements: continue, break and return.**

# Classes and methods

* The keyword **this** can be used for **unmasking** a variable of current class by others in a method(or **constructor**) and can be used to call other constructor in the same class
  + Must be used as first statement
* The **finalize( )** Method can be defined inside any class : executed before an object is claimed during garbage collection
* Methods and constructors can be overloaded
* Passing object to a method = call by reference
* Objects can be returned
* Recursion
* Any **static** member can be accessed before any object of the class is declared
* All instances of a class share same static variable, no copies of static variables are created
*  equivalent to *define int FILE\_NEW = 1; in c*
* Classes can be nested
* Class **string**: **length ()** and **equals ()**.
* **Variable arguments**: type …var\_name.

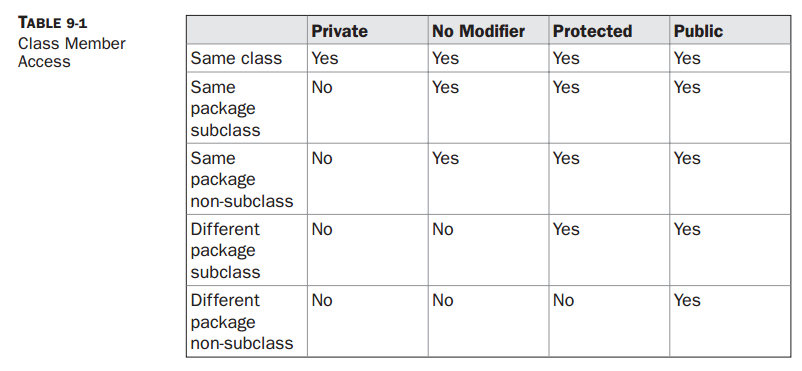
# Inheritance

* Class **extends** class; *class A{…} class B extends A{…}*
* Constructor **super()** is used as constructor to immediate superclass – should be the first statement in the class construtor; *class A{ A(){…} …} class B extends class A{int x; A{super();}…}*
* **Super.variable** can be used to unmask the superclass variables having same name as a subclass variable; *class A{int x;} class B extends A {int x; super.x//access super class variable x}*
* Super class object can reference subclass object but the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.
* method overriding : superclass object can call subclass method which has overridden superclass method through **dynamic method dispatch** - Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
* A class can be modified with keyword final to terminate its ability to be inherited.
* **Abstract class**: can have methods declared but not defined(such methods must be modified with abstract keyword). A class inheriting a abstract class should define the methods not defined but declared by superclass or itself be an abstract class
* All classes are subclasses of class **Object** offers following functions

# Packages and interfaces

* Packages are used to control the visibility of namespaces declared as below



* Access protection: 
* Importing a package or a class: **import pkg1[.pkg2].(classname|\*);**
* **Java.lang** is implicitly imported by compiler to all java classes.
* If two classes with same name are imported then then those classes must be fully qualified with package name.
* **Interface**: all variables are by default final, cannot have defined methods.
* Keyword **Implements** is used to implement a Interface

**Class classname[extends superclass] [implements interface [,interface...] ]**

* When a method in interface is implemented it must be declared public.
* Objects of type interface can refer any class that has the interface implemented.
* **Partial implementation:** If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**

**abstract class Incomplete implements Callback{}**

* Interface can be nested inside a class, doing so one has to qualify the interface with the class name e.g., class name implements clasname.interface.
* One interface can extend other interface.

# Exception handling

* Construction try {} catch(exceptiontype1 exob1){} catch (exceptiontype2 exob2){}
* All exception classes as subclasses of class **Throwable.** Two important subclasses are
  + **RuntimeException:** all exceptions created during runtime.
  + **Error:** errors directly linked to runtime environment.

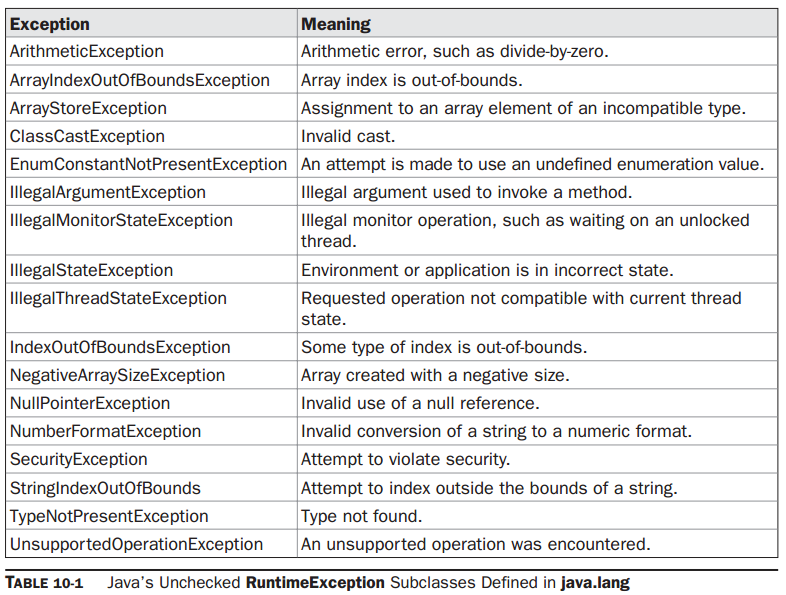
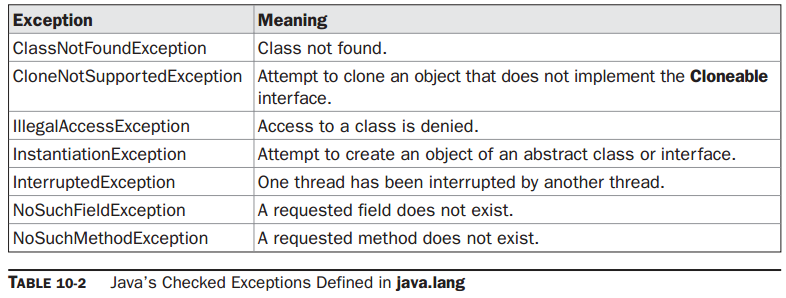
Both classes have default exception handlers.

* Try statements can be nested: Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement’s catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.
* **throw** can be used to throw an exception explicitly by the program.

**Throw ThrowableInstance;** // ThrowableInstance can be created using new keyword.

* **throws** can be usedforce the caller of the method handle the exception that may be created and not handled by the function.

E.g., type method-name (parameter-list) throwsexception-list {}

* The keyword finally used with try, whatever the way try block is exited, finally block will be executed.
* Built-in java exceptions: 
* 
* **Chained exceptions:**

{….// create an exception

NullPointerException e =new NullPointerException("top layer");

// add a cause

e.initCause(new ArithmeticException("cause"));

…..}

{ ….

System.out.println("Original cause: " +

e.getCause());

**ThrowableInstance.getCause()** returns underlying cause or null;

* **Creating your own exception:**

MyException(int a) {

detail = a;

}

static void compute(int a) throws MyException {

System.out.println("Called compute(" + a + ")");

if(a > 10)

throw new MyException(a);

…

catch (MyException e) {

System.out.println("Caught " + e);

# Enumeration and Autoboxing

* **Declaration** of enumeration : enum Apple {const1,const2,..} all members are public static by default.
* Enum constants in java are **class type**: they can be given constructor and methods (constructor and methods in given class is same to all classes) – yet no need to use new operator.
* All constants in java are **self-typed** (const1 is of type Apple – refer Declaration).

Apple ap; … ; ap=Apple.const1;

When using in switch: switch (ap) {case const1:… ; case const2: … ;}

* Enumeration.**values()** gives all constants in the given enumeration,

Apple allap[] = Apple.values(); // yield allap[]={cosnt1, const2, ..}

* Enumeration.**valueOf(str)** gives the value of str in the enumeration

, ap = Apple.valueOf(const1); //yields ap=const=1

* Cycling in foreach loop: for(Apple a : Apple.values()) {}
* When declaring enum, it is not possible to inherit a superclass
* All enum inherit java.lang.Enum which has three important methods
  + , final int ordinal: ap=Apple.const1; int I = ap.ordinal(); //yields i=1
  + , final int comparTo(): int i=ap.compareTo(ap1) //yields i=ap.ordinal-ap1.ordinal;
  + , Boolean equals(): boolean b = ap.equals(ap1); //yields b = true if type and ordinal of ap and ap1 matches.

Note: equals defined by object is overridden.

* There are classes corresponding to primitive types as below(shown as constructors in the first column) and methods in second column

Character (char ch) char charValue()

Boolean(Boolean boolval|string boolstring) boolean booleanValue()

(boolstring is true or false in upper case or lower case)

Integer(int num|string numvalue) int intValue( ) //numvalue must be a number

(constructors for following classes are of same same type above)

Byte byte byteValue( )

Double double doubleValue( )

Float float floatValue( )

Long long longValue( )

Short short shortValue( )

* Autoboxing and auto-unboxing occurs automatically in expressions.
* Primitive types are faster in excecution than their corresponding class type.

# Annotation

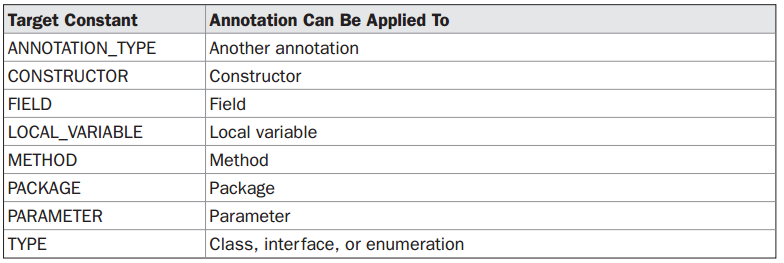
* Annotations are subclass of java.lang.annotation.
* **Declaration** of annotation: **@interface** anno{String str(); Integer val();} either or both the fields can have default value @interface anno{String str() default “msg”; Integer val();}
* **Attaching an annotation** to method @anno{str=”msg”; val=number;} method header and body;
* **Retention policy** : @Retention(RetentionPolicy.RUNTIME|SOURCE|CLASS)

Should be specified just before the annotation. Retention policy CLASS will result in annotation to be a .class file after compilation but will not be available for JVM.

* Method variable =object.getClass.getMetho() or getField or getConstructor is used to obtain the object which has been annotated;
* **Method getMethod(StringmethName, Class ...paramTypes)**
* **Annotation getAnnotation(Class annoType)** is used to get the annotation on given object.
* Displaying values : System.out.println(anno.str() + " " + anno.val());
* Getting all annotations : @interface anno{String str(); Integer val();}
* **Marker annotation** can be checked by isAnnotationPresent( ); Its sole purpose

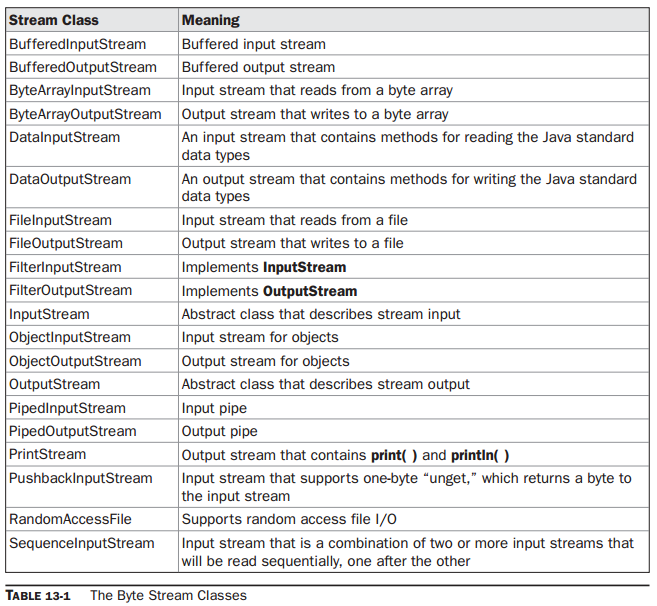
is to mark a declaration. e.g., m.isAnnotationPresent(MyMarker.class

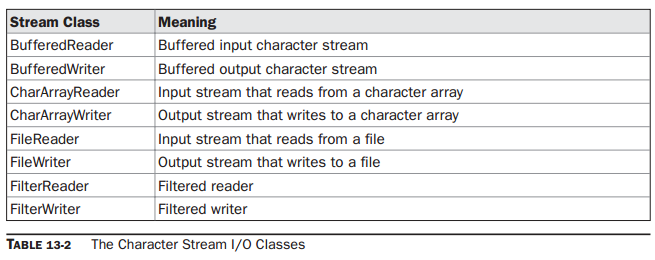
* **Single value** annotation : @interface single{Integer i();} @single(100) class singleannotated{}
* Java.lang.annotation has several inbuilt important classes as below
  + **@Retention** acts are annotation for other annotations specifying retention policies.
  + **@Documented** tells the tool, other annation must be documented. It is designed to be used only as an annotation to an annotation declaration.
  + **@Target** is used to specify the targets for the given annotation.it can take one parameter within (), of multiple parameters within ({here}), parameters are as follow,

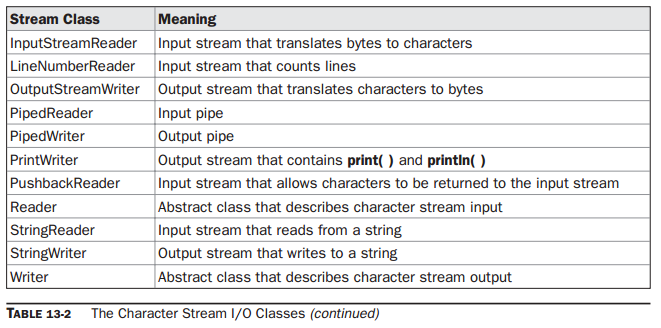


* + **@Inherited** can be used only on classes and it will be inherited to its subclasses.
  + **@Override** can be used only on methods, further this method must be overriding a superclass method.
  + **@SuppressWarnings** specifies that one or more warnings that might be issued by the compiler are to be suppressed. The warnings to suppress are specified by name, in string form. This annotation can beapplied to any type of declaration.
* Some restrictions
  + Annotation cannot inherit other.
  + Methods in annotations cannot have body and must return one of the following
    - A primitive type, such asintordouble
    - An object of type String or Class
    - An enum type
    - Another annotation type
    - An array of one of the preceding type
  + annotation methods cannot specify a throws clause.
  + Annotations cannot be generic.

# File handling







* Reading from console
  + BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  + int i=br.read( ) throws IOException;
  + String br.readLine();
* To write to console instead of println, write can be used , e.g., …write(int bytevalue);
* To read a file
  + FileInputStream fin = new FileInputStream(filename);
  + Int I =fin.read();
  + Fin.close();
* To write to a file
  + FileOutputStream fout = new FileOutputStream(filename); int I;
  + Fout.write(i);
  + Fout.close();

# Other topics

* **Transient modifier**: When an instance variable is declared astransient, then its value need not persist when an object is stored. For example:

class T {transient int a; // will not persist

int b; // will persist }

* **Volatile modifier**: instance variable modified by volatile will be constantly updated . (in multithread programs each thread may keep a copy of original variable).
* , **object instanceof typ** - will return true if object is of type typ or can be casted.
* **Stricfp** can be used as modifier for class or method to increase the precision of calculations.
* **Assert** has two forms
  + **, assert condition;**
  + **, assert condition:expression;**
  + if the condition in assertion fail, an exception is thrown, e.g.,

….

assert n > 0 : "n is negative!";

….

Exception in thread "main" java.lang.AssertionError: n is

negative!

* To run program with assertion: java –ea filename
* To enable and disable assertion in a package

-ea:MyPack

-da:MyPack

* **Static import :** if a method is imported as static, it can be used without qualification by its class name.

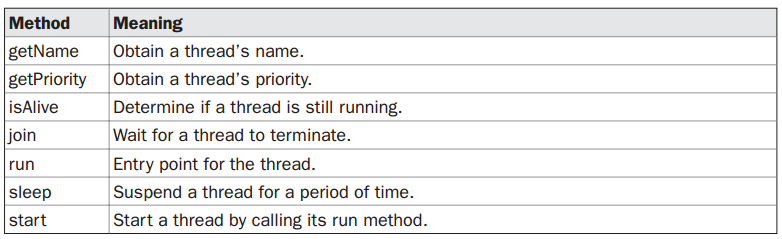
# Multithread programming

* Main thread should be the last to terminate.
* Current thread can be obtained by function **currentThread** (static function): Thread t= Thread,currentThread();
* …Println(t);This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is main. Its priority is 5, which is the default value, and mainis also the name of the group of threads to which this thread belongs. Athread groupis a data structure that controls the state of a collection of threads as a whole.

e.g., [main,5,main]

* Java’s multithreading system is built upon the **Thread class** (can be extended), its methods, and its companion interface **Runnable** (can be implemented and must implement its method **run()**).
* One of the method for Thread class : **Thread(RunnablethreadOb, StringthreadName)**
* For implementation of runnable: Thread t;

NewThread() {t = new Thread(this, "Demo Thread"); … }

* For Thread extended class: super(class name.)
* Few important methods in Thread class
* **final boolean isAlive( tob);**
* **final void join( tob) throws InterruptedException**
* **final void setPriority(int level)** //provides static final int MIN\_PRIORITY, MAX\_PRIORITY,NORM\_PRIORITY (1,10,5 respectively)
* access to a method can be synchronised by modification of method by keyword synchronized.

**Synchronized void call()**;

* access to a block through an object can be synchronized

**synchronized(object)** {

// statements to be synchronized

}

* **final void wait( )** throws InterruptedException //Additional forms ofwait( )exist that allow you to specify a period of time to wait
* **final void notify( )**
* **final void notifyAll( )**
* spurious wakeup: wakeup of a waiting thread without notified by method notify();
* **dead lock**: occurs when two threads have a circular dependency on a pair of synchronized

objects. suppose one thread enters the monitor on object X and another thread

enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y,

it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized

method on X, the thread waits forever, because to access X, it would have to release its own

lock on Y so that the first thread could complete.

Dead lock may involve more than 2 threads and 2 synchronised methods.

* Use a flag and periodic checking to suspend and resume a thread.

# Generics

* **Declaring** generic class: class Gen<T>{}
* Creating generic class **object**: Gen<Integer> iOb = new Gen<Integer>(8);
* **Wildcards** (?) specially useful when using generic class object as argument for a function.

boolean sameAvg(Stats<?> ob){}

* Upper bound of generic parameter including wildcards can defined using **extends** keyword.
* Lower bound to a wildcard can be defined by **super** keyword.
* **Generic methods or constructors**: <t extends Number> genmc(t a){}
* **Generic interfaces:** interface MinMax<T extends Comparable<T>>

interface MinMax<T extends Comparable<T>> //correct

class MyClass implements MinMax<T> { // Wrong!

class MyClass implements MinMax<Integer> { // OK

* Class inheriting generic supeclass must also be generic, specifying atleast the generic parameters in the superclass;
* **Instanceof** works with generic classes as well.//wildcards are useful here.
* **Casting:** You can cast one instance of a generic class into another only if the two are otherwise compatible and their type arguments are the same.

(Gen<Integer>) iOb2

* **Erasure:** When your Java code is compiled, all generic type information is removed (erased). This means replacing type parameters with their bound type, which isObjectif no explicit bound is specified**.**
* **Bridge method.**
* **Ambiguity errors;**
* **Some restrictions:**
  + Type Parameters Can’t Be Instantiated
  + No static member can use a type parameter declared by the enclosing class
  + you cannot instantiate an array whose base type is a type parameter.
  + you cannot create an array of type-specific generic references.