**Source File Declaration Rules:**

* One public class per source code file
* If there is a public class in the file, the name of the file must match the public class name
* If the class is the part of a package, the package statement should be the first line of the source code before any import statements.’
* If there are import statements, they must go between the package name (if there is one) and class declaration

**Using The javac and java commands:**

**Compiling With Javac:**

The javac command is used to invoke java’s compiler. You can specify many options when running javac.

**And what are those options like:**

For instances, there are options to generate debugging information or compiler warning.

javac [options] [source-files]

**Some of the examples of javac command:**

**Javac -help**

**Javac -version foo.java Bar.java**

The first invocation does not compile any files, but prints a summary of valid options.

The second invocation passes the compiler an option, (-version, which prints the version of the compiler they are using) and passes the compiler two java files to compile, (foo.java and bar.java). Whenever, you are specifying multiple options, they must be separated by spaces.

**Launching Applications With Java: java command:**

The java command is used to invoke the java virtual machine.

**Static Import:**for instance, consider the following sample program:

**import static java.lang.System.out;**

**import static java.lang.Integer.\*;**

**public class TestStaticImport**

**{**

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

**{**

**out.println(MAX\_VALUE);**

**out.println(toHexString(42));**

**}**

**}**

Let’s look what’s happening in the code that’s using the static import.

* Even through the feature is static import, the syntax must be import static followed by the fully qualified name of the static member you want to import. Or, you could use wildcard. (**like, import static java.lang.Integer.\*; here, \* is the wildcard)**
* What does static import actually mean? For instance, **import static java.lang.Integer.\*;** it says I want to do static imports of all the static members.
* Now, we are fully seeing the benefits of static import features. First, it imports all the static members, second, We did not have to type System in System.out.println. Second thing, we don’t need to type the Integer in Integer.MAX\_VALUE. So, in this line, we were able to use a short cut for a static\_method and a constant.

(However, in the last case, where we don’t have to import the integer in Integer.MAX\_VALUE, watch out for the ambiguously named static members. For instance, if your program does a static import for both the classes Integer and Long, referring to the MAX\_VALUE will cause a compiler error. Since, for both classes static definition of MAX\_VALUE is present and compiler does not know which MAX\_VALUE you are referring.

**Wildcard Concept In import:**

As you have seen, when using import and import static statements, sometimes you can use the wildcard character \* to do the simple searching (for a function or constant) for you. **(with the use of \* character, you can search through a package or within a class). you can say this:**

**import java.util.\*; //ok, to search the whole java.util packages**

In a similar vein, if you want to search the java.lang.Integer class for static members, you can say that:

**import static java.lang.integer.\*;**

But, you cannot create broader searches. For instance, you cannot use an import to search through the entire java API;

**import java.\*;**

**Class Declarations And Modifiers:**

Class modifiers are fall into two types:

**Access Modifiers. (public, private, protected)**

**Non access modifiers. (strictfp, final and abstract)**

**Access Modifiers For A Class:**Now, though there is three access modifiers, public, private and protected, there are four levels of access controls. As the fourth one is default or package access when you don’t use any of the chosen access modifiers. (private, protected, public)

However, all the four access modifiers are for class variables and functions. For a class, there is two access modifiers. **Public and Default**

(java is a package centric language, the developers assumed that for **good organization and for named scoping**, you would put all your classes into packages. And, this is right, Otherwise, consider the following situation. Three different programmers which are in the same company but working on different projects,define their own utilities class. Now, if those classes are not declared in any of the explicit package, and are in the class path, there is no way to tell JVM or compiler which one you are trying to reference.

Oracle suggests/recommends that developer use **reverse domain names appended with division and/or project names. For example, if your domain name is anonymous.com your package name should start with com.anonymous.**

)

**Class Access:**

**Default and public access.**

Now,

There are four modifiers which can be used for class. Public, protected, default, private.

**Public access modifier:**

When a method or variable member is declared public, it means all other classes, regardless of the package they belong to, can access the member. (Assuming that the class itself is visible)

**package cert;**

**public class sludge**

**{**

**public void testit(){System.out.println(“Sludge”)};**

**}**

**package book;**

**import cert.\*;  
class Goo**

**{**

**Public static void main(String args[])**

**{**

**sludge o=new sludge();**

**O.testIt();**

**}**

**}**

As you can see, Goo and sludge are in different packages. However, Goo can invoke the method in sludge without problems, because, both the sludge class and it’s testIt() method are made public.

But, if the sludge class is not public, it will not happen.

**Private access modifiers:**

Members marked private cannot be accessed by code in other class other than the class in which the private member is declared.   
  
  
**Now, Note that, a private method of a super class cannot be overridden by a subclass since, it is not inheriting it.**

**Protected Access modifiers:**

The protected and default access control levels are almost identical. But, with one critical difference. A default member may be accessed only if the class accessing the member belongs to thee same package. Whereas, a protected member can be accessed (through inheritance) by a subclass if the subclass is in a different package.

**Any class could only see the protected members through inheritance.**

**Non Access Modifiers Of A Class:**

strictfp, final and abstract. (what is native? Native is a modifier used on a function name. But it finally uses the native interface to reuse the functions defined in other languages)

**Final Class:**when used in a class declaration, the final keyword means the class cannot be sub classed. In other words, no other class can ever extend a final class. And trying to do so will generate **Compilation error. In other words, no other class can ever extend.** You should make a final class only if you need an absolute guarantee that none of the methods in that class will ever be overridden.

**Many classes in java core libraries are final. Like, String class.** Imagine the havoc if you could not guarantee how a string object would work on any given system your application is running on.

**However, in practice, we will almost never make a final class.** A final class obliterates a key benefit of OO -extensibility. So, unless if you have a serious safety or security issue, you should not do this.

Now, final keyword can be used for methods, too. In case it is used for methods, it prevents a method from being overridden in a subclass.

Final keyword can also be used in case of an argument. The concept is similar to const argument in c++. **A new value cannot be assigned to the variable. If that is passed ass a final argument.**

**Abstract Class:**

An abstract class can never be instantiated. So, it’s purpose is it has to be extended.

**So, conceptually where it is useful?**

Imagine you have a class car that has generic methods common to all vehicles. But, you don’t want anyone to actually create a generic, abstract class object.

Or, the bank account example.

**abstract class Car**

**{**

**private double Price;**

**private String model;**

**private String year;**

**private abstract void goFast();**

**private abstract void goUpHill();**

**private abstract void impressNeighbours();**

**}**

**Some points about abstract class:**

* Even a single method is abstract in a normal class, that class has to be defined as abstract.
* However, you can have non abstract methods in a abstract class. For example, you might have methods that should not change from Car Type to Car Type such as getColor() or setPrice(). By putting non-abstract methods in a abstract class, you give all concrete subclasses inherited method implementation.

**Strictfp modifier:**

**strictfp** is a keyword in java used for restricting floating-point calculations and ensuring same result on every platform while performing operations in the floating-point variable.  
Floating point calculations are platform dependent i.e. different output(floating-point values) is achieved when a class file is run on different platforms(16/32/64 bit processors). To solve this types of issue, strictfp keyword was introduced in JDK 1.2 version by following [IEEE 754](https://en.wikipedia.org/wiki/IEEE_floating_point) standards for floating-point calculations.

**Important points:**

* strictfp modifier is used with classes, interfaces and methods only.

**strictfp class Test**

**{**

**// all concrete methods here are**

**// implicitly strictfp.**

**}**

**strictfp interface Test**

**{**

**// all methods here becomes implicitly**

**// strictfp when used during inheritance.**

**}**

**class Car**

**{**

**// strictfp applied on a concrete method**

**strictfp void calculateSpeed(){}**

**}**

* When a class or an interface is declared with strictfp modifier, then all methods declared in the class/interface, and all nested types declared in the class, are implicitly strictfp.
* strictfp cannot be used with abstract methods. However, it can be used with abstract classes/interfaces. **(since, only concrete methods can be strictfp)**
* Since methods of an interface are implicitly abstract, strictfp cannot be used with any method inside an interface.

**strictfp interface Test**

**{**

**double sum();**

**strictfp double mul(); // compile-time error here**

**}**

//Java program to illustrate strictfp modifier

**public class Test**

**{**

**// calculating sum using strictfp modifier**

**public strictfp double sum()**

**{**

**double num1 = 10e+10;**

**double num2 = 6e+08;**

**return (num1+num2);**

**}**

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

**{**

**Test t = new Test();**

**System.out.println(t.sum());**

**}**

**}**

**User Interfaces:**When you create an interface, you are defining what a class can do without saying how the class will do it.

Any class type that implements the interface must write code for all methods.

(Difference between interface and abstract class:

Now, abstract class-inheritance-**is a relationship, Whereas, modular kitchen is implemented by normal kitchen)**

**However, except the theoretical part, technically consider interface as a 100% abstract class)**

**Now, some points for abstract class:**

* All interface methods are implicitly public and abstract. In other words, you do not need to type public or abstract modifiers in the method declaration, but the method will always be public and abstract.
* All variables defined in an interface must be public, static and final. **(and these modifiers are not implicit. You have to mention it).** In other words, **interfaces can only have constants, not instance variables.**
* However, unlike, the variables, which should be **public, static and final,** interfaces method should not be static.
* Because, interface methods are abstract, they cannot be marked as final.
* **An interface can extend one or more other interfaces.**
* An interface cannot extend anything but another interface.
* **An interface cannot implements another interface or class.**
* An interface must be declared with the keyword interface  
    
  Consider the following example:

**public interface bouncable**

**{**

**void bounce();**

**void setbouncefactor(int bf);**

**}**

* Because, interface methods are abstract (implicitly), we cannot use **final, strictfp or native modifiers with them.**
* Interface types can be used polymorphically.
* **Further Note:** the following is a legal interface declaration:

**public abstract interface Rollable()**

**{**

**//the variables (which must be static, public and final)**

**//the methods (which are implicitly public and abstract)**

**}**

However, typing in the **abstract modifier** is considered redundant.

# native keyword in java and native interface:

The native keyword is applied to a method to indicates that the method is implemented in native code using JNI (Java Native Interface). native is a modifier applicable **only for method**s and we can’t apply it anywhere else. The methods which are implemented in C, C++ are called as native methods or foreign methods.

**The main objective of native keyword are:**

* To improve performance of the system.
* To achieve mission level/memory level communication.
* To use already existing legacy non-java code.

**Pseudo code to use native keyword in java:**

|  |
| --- |
| **Class Native**  **{**  **Static**  **{**  **System.LoadLibrary(“Native library path”);**  **}**  **Public native void m();**  **}**  **Class Test**  **{**  **Public static void main(String[] args)**  **{**  **Native n = new Native();**  **n.m();**  **}**  **}** |

**Important points about native keyword:**

* For native methods implementation is already available in old languages like C, C++ and we are not responsible to provide implementation. Hence native method declaration should ends with ; (semi-colon). **(since, we are not responsible to provide implementation. Hence, we only need to declare a function)**
* We can’t declare native method as [abstract](https://www.geeksforgeeks.org/abstract-classes-in-java/).
* We can’t declare native method as [strictfp](https://www.geeksforgeeks.org/strictfp-keyword-java/) because there is no guarantee that old languages (C, C++) follow IEEE 754 standard. Hence native **strictfp** combination is illegal combination for methods.
* The main advantage of native keyword is performance will be improved but the main disadvantage of native keyword is it breaks platform independent nature of java.

A class cannot inherit multiple classes in java, however, is free to extend multiple interfaces.

An interface can extends multiple interface.

An interface can not implement another interface.

**Methods With Variable Argument Lists:**

**void dostuff(int …x)**

**{**

**}**

Remember, the three dots

Expects from 0 to as many ints as parameters.

**void dostufff2(char c,int …x)**

**{**

**}**

This expects a first char, then 0 to many ints.

This syntax tells the compiler that fun( ) can be called with zero or more arguments. As a result, here a is implicitly declared as an array of type int[]. Below is a code snippet for illustrating the above concept :

**// Java program to demonstrate varargs**

**class Test1**

**{**

**// A method that takes variable number of intger**

**// arguments.**

**static void fun(int ...a)**

**{**

**System.out.println("Number of arguments: " + a.length);**

**// using for each loop to display contents of a**

**for (int i: a)**

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

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

**}**

**// Driver code**

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

**{**

**// Calling the varargs method with different number**

**// of parameters**

**fun(100); // one parameter**

**fun(1, 2, 3, 4); // four parameters**

**fun(); // no parameter**

**}**

**}**

Another example:

**// Java program to demonstrate varargs with normal**

**// arguments**

**class Test2**

**{**

**// Takes string as a argument followed by varargs**

**static void fun2(String str, int ...a)**

**{**

**System.out.println("String: " + str);**

**System.out.println("Number of arguments is: "+ a.length);**

**// using for each loop to display contents of a**

**for (int i: a)**

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

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

**}**

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

**{**

**// Calling fun2() with different parameter**

**fun2("GeeksforGeeks", 100, 200);**

**fun2("CSPortal", 1, 2, 3, 4, 5);**

**fun2("forGeeks");**

**}**

**}**

Now, remember, there cannot be two varargs which belong to the same function.

**void method(String... gfg, int... q)**

**{**

**// Compile time error as there are two**

**// varargs**

**}**

Also, varargs must be the last parameter of a functions, if there is more than one parameter to the function.

**void method(int... gfg, String q)**

**{**

**// Compile time error as vararg appear**

**// before normal argument**

**}**

**Variable Declarations:**there are two types of variables in java.   
  
**Primitives:** a **primitive** can be one of the eight types. Char, boolean, short, int, long, double or float. Once, a primitive has been declared, its primitive type can never be changed. Although in most cases, its value can be changed.

**Reference Variables:** a reference variable is used to refer to an object. A reference variable is declared to be a specific type and that type can never be changed. A reference variable can be used to refer to any other objects of the declared type or of a subtype of the declared type.

**A Nice Thing About Java Reference:   
  
If Java uses the pass-by reference, why won't a swap function work?**  
A: Java does manipulate objects by reference, and all object variables are references. However, Java doesn't pass method arguments by reference; it passes them by value.  
  
Take the badSwap() method for example:  
  
public void badSwap(int var1, int var2)  
{  
 int temp = var1;  
 var1 = var2;  
 var2 = temp;  
}  
When badSwap() returns, the variables passed as arguments will still hold their original values. The method will also fail if we change the arguments type from int to Object, since Java passes object references by value as well.

so , what does that mean?

**import java.util.Vector;**

**public class Test**

**{**

**public static void addInContainer(Vector<Integer> container)**

**{**

**container.add(20);**

**container.add(30);**

**}**

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

**{**

**Vector<Integer> container=new Vector<>();**

**container.add(10);**

**//Now, print the values**

**for(int i=0;i<container.size();i++)**

**{**

**System.out.print(container.elementAt(i)+" ");**

**}**

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

**addInContainer(container);**

**System.out.println("After calling addInContainer");**

**for(int i=0;i<container.size();i++)**

**{**

**System.out.print(container.elementAt(i)+" ");**

**}**

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

**}**

**}**

**This will print:**

10   
After calling addInContainer  
10 20 30

**import java.util.Vector;**

**public class Test**

**{**

**public void swap(Vector<Integer> container)**

**{**

**Vector<Integer> container2=new Vector<Integer>();**

**container=container2;**

**}**

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

**{**

**Vector<Integer> container=new Vector<>();**

**container.add(10);**

**container.add(20);**

**container.add(30);**

**//Now, print the values**

**for(int i=0;i<container.size();i++)**

**{**

**System.out.print(container.elementAt(i)+" ");**

**}**

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

**System.out.println("After swap function is called");**

**for(int i=0;i<container.size();i++)**

**{**

**System.out.print(container.elementAt(i)+" ");**

**}**

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

**}**

**}**

**This will print:**

10 20 30   
After swap function is called  
10 20 30

**Declaring Reference Variables:**Reference variables can be declared as static, instance (local to a class and not static.), function parameters or local variables.

**Local Variables:**

A local variable can only be final.

**Instance Variables:**apart of any of the of the four access levels (by using four access specifiers: public, private, protected, default) an instance variable can be marked as the following:  
  
Can be marked final

Can be marked transient. (it has importance. It prevents user’s private yet unnecessary data to be saved in a file)

Can be marked volatile. (if multiple threads access a sharable object in a multiprocessor system, they have their own copy of the sharable in the cache. To prevent that, this is used)

Cannot be marked as abstract.

Cannot be synchronized.

Cannot be marked as strictfp.

Cannot be marked native.

Cannot be marked as static. Because, after marking it with static, they would become the class variable.

**Now, a variable cannot be marked as abstract. A class or function is.**

**Now, generally synchronized is not used with instance variables.** However, synchronized is used, in other cases. It can be used with methods, too.

strictfp modifier is used with classes, interfaces and methods only. (also, it can be applied to concrete methods, not abstract methods)

Native is generally used with functions/methods. It cannot be use with an instance variable.

**Now, we already know what is the meaning of a variable when it is marked as final.**

**Transient Keyword:**

transient is a variables modifier used in serialization. At the time of serialization, if we don’t want to save value of a particular variable in a file, then we use transient keyword. When JVM comes across transient keyword, it ignores original value of the variable and save default value of that variable data type.

transient keyword plays an important role to meet security constraints. **There are various real-life examples where we don’t want to save private data in file.** Another use of transient keyword is not to serialize the variable whose value can be calculated/derived using other serialized objects or system such as age of a person, current date, etc.’

(Now, before we see an example, let’s understand the basics of serialize and deserealize.

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement the java.io.Serializable interface.

)

Now, consider the following example:

**import java.io.Serializable;**

**public class Student implements Serializable**

**{**

**int id;**

**String name;**

**transient int age;//Now it will not be serialized**

**public Student(int id, String name,int age)**

**{**

**this.id = id;**

**this.name = name;**

**this.age=age;**

**}**

**}**

Now write the code to serialize the object.

**import java.io.\*;**

**class PersistExample**

**{**

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

**Student s1 =new Student(211,"ravi",22);//creating object**

**//writing object into file**

**FileOutputStream f=new FileOutputStream("f.txt");**

**ObjectOutputStream out=new ObjectOutputStream(f);**

**out.writeObject(s1);**

**out.flush();**

**out.close();**

**f.close();**

**System.out.println("success");**

**}**

**}**

Now, A FileOutoutStream is an file output stream, which is associated with f.txt file. Now, we have converted the object s1 to a ByteStream and have written it into a file.

Now write the code for deserialization.

**import java.io.\*;**

**class DePersist{**

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

**ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));**

**Student s=(Student)in.readObject();**

**System.out.println(s.id+" "+s.name+" "+s.age);**

**in.close();**

**}**

**}**

The output will be:

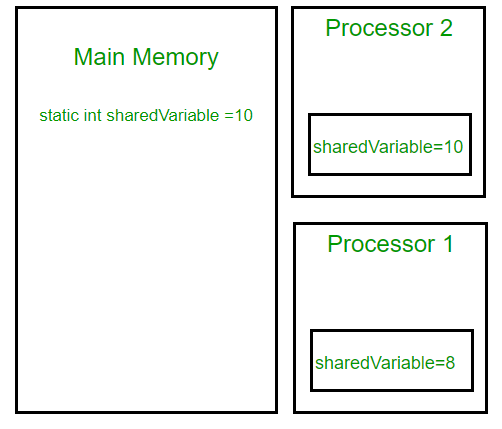
211 ravi 0

As you can see, printing age of the student returns 0 because value of age was not serialized. Because, the variables is not serialized.

This way, we could prevent unnecessary yet user’s private data to be saved in the file.

**Volatile Variable:**  
Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe. Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

Suppose that two threads are working on SharedObj. If two threads run on different processors each thread may have its own local copy of sharedVariable. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the write policy of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.



Although most modern hardware provide good cache coherence therefore most probably the changes in one cache are reflected in other but it’s not a good practice to rely on hardware for to ‘fix’ a faulty application.

Note that volatile should not be confused with static modifier. **static variables are class members that are shared among all objects. There is only one copy of them in main memory.**

**Whereas, the for volatile class members, each thread can have it’s own copy of sharable objects (in cache) if they are running on different processors(all modern day computers have multi core processors)**

**volatile vs synchronized:**

Before we move on let’s take a look at two important features of locks and synchronization.

Mutual Exclusion: It means that only one thread or process can execute a block of code (critical section) at a time.

Visibility: It means that changes made by one thread to shared data are visible to other threads.

Java’s synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modifies the value of shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other thread trying to enter the block at the same time will be blocked and put to sleep.

In some cases we may only desire the visibility and not atomicity. Use of synchronized in such situation is an overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features. The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory. However, use of volatile is limited to very restricted set of cases as most of the times atomicity is desired. For example a simple increment statement such as x = x + 1; or x++ seems to be a single operation but is s really a compound read-modify-write sequence of operations that must execute atomically.

**The Java volatile Happens-Before Guarantee**

To address the instruction reordering challenge, the Java volatile keyword gives a "happens-before" guarantee, in addition to the visibility guarantee. The happens-before guarantee guarantees that:

Reads from and writes to other variables cannot be reordered to occur after a write to a volatile variable, if the reads / writes originally occurred before the write to the volatile variable.

The reads / writes before a write to a volatile variable are guaranteed to "happen before" the write to the volatile variable. Notice that it is still possible for e.g. reads / writes of other variables located after a write to a volatile to be reordered to occur before that write to the volatile. Just not the other way around. From after to before is allowed, but from before to after is not allowed.

Reads from and writes to other variables cannot be reordered to occur before a read of a volatile variable, if the reads / writes originally occurred after the read of the volatile variable. Notice that it is possible for reads of other variables that occur before the read of a volatile variable can be reordered to occur after the read of the volatile. Just not the other way around. From before to after is allowed, but from after to before is not allowed.

The above happens-before guarantee assures that the visibility guarantee of the volatile keyword are being enforced.

So, what does that mean? **That means compiler cannot optimize the code when volatile variable is there.**

**Enum:**

As of java 5, java lets you restrict a variable to have one of the predefined values. In other words, one value from an enumerated list.

Using enum can help in reducing the bugs in your code.

For instance, in your coffee shop application, you might want to restrict your coffee size size selections to BIG, HUGE, and OVERWHELMING. If you let and order for a **LARGE** or **GRANDE slip in**, it might cause an error.   
  
An enum can be defined as the following:

enum coffeesize={BIG, HUGE, OVERWHEELMING};

It’s not required that enum constants be all in caps, but borrowing from the Oracle code conventions, that constants are named in caps, it’s a good idea.

**Now, enum could be declared out side of a class, it could be declared within a class as a class member, or enum can be declared as their own separate class.**

**enum CoffeeSize{BIG, HUGE, OVERWHELMING}**

**//no semicolon at the end**

**//this cannot be private or protected**

**class Coffee**

**{**

**CoffeeSize size;**

**}**

**public class CoffeeTest1**

**{**

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

**{**

**Coffee drink=new Coffee();**

**drink.size=CoffeeSize.BIG;**

**System.out.println("The drink size is: "+drink.size);**

**}**

**}**

Now, it is ok. Now, note that, both classes are in same package, that’s why we are able to initialize it like that, drink.size=CoffeeSize.BIG; (because, it is default access modifier)

Now, as we mentioned previously, an enum can be declared within the class.

**class Coffee**

**{**

**enum CoffeeSize{BIG, HUGE, OVERWHELMING}**

**CoffeeSize size;**

**}**

**public class CoffeeTest1**

**{**

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

**{**

**Coffee drink=new Coffee();**

**drink.size=Coffee.CoffeeSize.BIG;**

**System.out.println("The drink size is: "+drink.size);**

**}**

**}**

But, note that, an enum cannot be declared within the method.

Now, note that thing. Java language designers make it optional to put a semicolon at the end of the enum declaration.

So, what gets created when you make an enum? The most important thing to remember that an enum is not string or int. Each of the enumerated CoffeeSize types **are actually an instance of CoffeeSize. Think of an enum as a kind of class that looks something like this (not exactly though)**

**class CoffeeSize**

**{**

**public static final CoffeeSize BIG=new CoffeeSize(“BIG”,0);**

**public static final CoffeeSize HUGE=new CoffeeSize(“HUGE”,1);**

**public static final CoffeeSize OVERWHELMING=new CoffeeSize(“OVERWHELMING”,”2”);**

**CoffeeSize(String enumName, int index)**

**{**

**//stuff here**

**}**

**}**

**(**

**java valueOf() method:**

The valueOf method returns the relevant Number Object holding the value of the argument passed. The argument can be a primitive data type, String, etc.

**Following are all the variants of this method −**

static Integer valueOf(int i)

static Integer valueOf(String s)

static Integer valueOf(String s, int radix)

**Parameters**

Here is the detail of parameters −

**i −** An int for which Integer representation would be returned.

**s −** A String for which Integer representation would be returned.

**radix −** This would be used to decide the value of returned Integer based on the passed String.

**Return Value**

**valueOf(int i) −** This returns an Integer object holding the value of the specified primitive.

**valueOf(String s) −** This returns an Integer object holding the value of the specified string representation.

**valueOf(String s, int radix) −** This returns an Integer object holding the integer value of the specified string representation, parsed with the value of radix. (this radix value is actually the base in which the value will be printed. Like, base of 2, base of 10, base of 8, base of 16)

**Example**

**public class Test {**

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

**Integer x =Integer.valueOf(9);**

**Double c = Double.valueOf(5);**

**Float a = Float.valueOf("80");**

**Integer b = Integer.valueOf("444",16);**

**System.out.println(x);**

**System.out.println(c);**

**System.out.println(a);**

**System.out.println(b);**

**}**

**}**

This will produce the following result −

**Output:**

9

5.0

80.0

1092

)

Java programming language enum types are much more powerful than their counterparts in other languages. The enum declaration defines a class (called an enum type). The enum class body can include methods and other fields.

In order to see the actual size of each enum, let's make an actual enum and examine the contents of the class file it creates.

Let's say we have the following Constants enum class:

**public enum Constants {**

**ONE,**

**TWO,**

**THREE;**

**}**

Compiling the above enum and disassembling resulting class file with javap gives the following: (Now, javap disasssmbles the machine code generatedd, as disassembling **(Compiling the above enum and disassembling resulting class file with javap gives the following)**

**Compiled from "Constants.java"**

public final class Constants extends java.lang.Enum{

public static final Constants ONE;

public static final Constants TWO;

public static final Constants THREE;

public static Constants[] values();

public static Constants valueOf(java.lang.String);

static {};

}

The disassembly shows that that each field of an enum is an instance of the Constants enum class. (Further analysis with javap will reveal that each field is initialized by creating a new object by calling the new Constants(String) constructor in the static initialization block.)

Therefore, we can tell that each enum field that we create will be at least as much as the overhead of creating an object in the JVM.

**Declaring Constructors, Methods, Variables In An Enum:**

Because, enum is a special kind of class, you can do more than just list the enumerated constant values. You can add constructors, instance variables, methods and something really strange known as a constant specific class body. To understand, why you might need more in your enum, think about the particular scenario: **imagine you want to know the actual size, in ounces, that map to the three CoffeeSize constants. Now, you could make some kind of lookup table using some other data structures. But that will be a poor design and hard to maintain. The simplest way to treat your enum values as objects, each of which can have its own instance variables and own values.**

**enum CoffeeSize**

**{**

**BIG(8), HUGE(10), OVERWHELMING(16);**

**private int ounces;**

**CoffeeSize(int ounces)**

**{**

**this.ounces=ounces;**

**}**

**public int getOunces()**

**{**

**return ounces;**

**}**

**}**

**public class Coffee**

**{**

**CoffeeSize size;**

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

**{**

**Coffee drink1=new Coffee();**

**drink1.size=CoffeeSize.BIG;**

**System.out.println("In "+drink1.size+" we get "+drink1.size.getOunces());**

**}**

**}**

Which produces: **In BIG we get 8**

**There are some points to know:**

* You can never invoke an enum constructor directly. The enum constructor is invoked automatically, with the arguments you defined after the constant value.
* You can define more than one argument to the constructor, and you can overload the enum constructors. Just as you overload a normal class constructor.

**Programming Paradigm, OOP:**

**What is Programming Paradigm?**

Answer) Programming paradigm is a fundamental style of computer programming. It is a way of building the structures and functions around the program.

**Object Oriented Programming Paradigm:**

**Answer)**

OOP treats data as a critical element and does not allow data to flow freely around the system.

OOP ties data close to the function which operates on it

OOP allows decomposition of a function into a set of entities called objects(runtime instance of class) and builds data data around them

The data of an object can be accessed only by the function associated with the objects.

One object can communicate with other objects using function of that object.

**Describe four key concepts of object oriented programming.**

Answer) The four key concepts of object oriented programming is 1) Encapsulation 2) Abstraction 3) Inheritance 4) Polymorphism

**1) Encapsulation:** The wrapping up of data and methods into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world. Only those methods, which are wrapped in the class, can access it.

So, these methods (which are wrapped in the class) provides the object's data and the program. This insulation of the data from direct access by the program is called 'data hiding'. (Private data members are example of data encapsulation)

**2) Abstraction:** Abstraction refers to the act of representing essential features of a class and omit the unnecessary details about it. For instance, when we think about a car, we don't consider all the irrelevant details like how the break actually works or how the Bluetooth device is installed in the

car, how it provides all the cool features. We think it as a transport medium which could take us from one place to another place. That is abstraction.

Now, the difference between abstraction and encapsulation:

One is mechanism hiding and another is data hiding.

**3) Inheritance:** Inheritance is a way by which a newly defined class inherits attributes and behaviour of an existing class along with its own properties.

Using inheritance the hierarchical relationships are established.

Inheritance allows the re usability of an existing operations and extending the basic unit of the a class without creating it from the scratch.

**Some more points about Inheritance:**

Inheritance is a “is a” relationship (not a “has a” relationship)

Like: we can say “A four-wheeler is a car” (It's a real “is a” relationship)

But, we cannot say, “A steering wheel is a car” (It's a “has a” relationship)

(A has a relationship is used in composition. An example of composition: A structure with in a

structure as a member)

Inheritance promotes both. Polymorphism and code re usability.

**4) Polymorphism:** Polymorphism is sharing a common interface for multiple types but having different implementation for different types.

In OOP, polymorphism is a technique where objects of classes belonging to

the same hierarchical tree may posses interface bearing the same name but each having different behaviors.

It is the way of inheriting when useful, overriding when not useful.

It allows automatically do the current behaviour even if we are working with

many different forms

**Define class.**

Answer) Class is a static definition of new type as a collection of data and associated operations from which runtime instances called objects can be created.

**Define object.**

Answer) Object is runtime instance of a conceptual framework encapsulating typed data and typed operations that correspond to a real world entity or thing for the purpose of computer modeling.

**Note:** A class cannot extend more than one class that means more than one parent per class. So, unlike, c++, java does not support multiple inheritance

**Overriding/Overloading:  
  
Overriding:**

Any time a class inherits a method form a super class, you have the opportunity to override the method**(unless, as you learned earlier, the method is final)**

The key benefit of overriding is to define the ability to define behaviour that’s specific to a particular subclass type.

For abstract methods you inherit from a superclass, you have no choice: but to implement the method in the subclass unless the subclass is also abstract. **(unlike c++, here in case of abstract class, abstract keyword is to be mentioned explicitly)**

**However, polymorphism, in which a base class object holds the reference of subclass (it can be done based on the reference rule) is a tricky one.**

**In that case, the base class object, cannot call the functions which are only specific to the subclass , whose reference it is currently holding.**

**class Animal**

**{**

**public void eat()**

**{**

**System.out.println("Generic animal eating generically");**

**}**

**}**

**class Horse extends Animal**

**{**

**public void eat()**

**{**

**System.out.println("Horse eating hay, oats and horse treats");**

**}**

**public void buck()**

**{**

**}**

**}**

**public class PolymorphismTrial**

**{**

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

**{**

**Animal c=new Horse();**

**c.buck();**

**}**

**}**

This will generate a compilation error.

**Another important point** is overriding method cannot have a more restrictive access modifier than the method being overridden.

For instance, the following code will throw compilation error:

**class Animal**

**{**

**public void eat()**

**{**

**System.out.println("Generic animal eating generically");**

**}**

**}**

**class Horse extends Animal**

**{**

**private void eat()**

**{**

**System.out.println("Horse eating hay, oats and horse treats");**

**}**

**}**

**public class TestAnimals**

**{**

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

**{**

**Animal a=new Animal();**

**Animal b=new Horse();**

**a.eat();**

**b.eat();**

**}**

**}**

TestAnimals.java:10: error: eat() in Horse cannot override eat() in Animal

private void eat()

^

attempting to assign weaker access privileges; was public

1 error

**The basic overriding rules are the following:**

* The argument list must exactly match that of the overridden method, If they don’t match, you can end up with an overloaded method you did not intend.
* The return type must be same as, or a subtype of, the return type declared in in the original method in the superclass
* The access level of a overridden function cannot be more restricted than the original method.
* Instance methods can only be overridden only if they are inherited by the subclass.
* The overriding method can throw any unchecked exception, regardless of whether the original method declares the exception
* The overriding method must not throw any checked exception that are new or broader than those declared by the original method. For example, a method which declares a FileNotFoundException, cannot be overridden by a method which throws SQLException, Exception or any other runtime exception unless it is subclass of **FileNotFoundException**
* The overriding method can throw narrower and/or fewer exceptions.
* You cannot override a method marked as final.
* You cannot override a method that is a static. (because, static methods are class specific, not instance specific)

**Invoking A SuperClass Version of the Overridden Method:**

That can be done using super keyword. Suppose, a subclass overrides the method eat() of a superclass, it can invoked the super class’s method by using

**super** keyword.

**super.eat()**

Now, the standard rules are obviously followed of **method overriding.**

However, there is an additional thing.

You can use super only to access a method in a class’s superclass, not the superclass of the superclass. That is you cannot say **super.super.doStuff()**

If a method is overridden, but you use a polymorphic(supertype) reference to refer

To the subtype object with the overriding method, the **compiler (compile time)**  assumes you are calling the supertype version of the method. If the supertype version declares a checked exception, but the overriding subtype method does not, the **compiler** still thinks you are calling a method that declares an exception.

**Overloaded Methods:**

* overloaded method must change the argument list.
* Overloaded method can change the return type.
* Overloaded method can change the access modifier.
* Overloaded methods can declare new or broader checked exception.
* A method can be overloaded in the same class or subclass. (however,the first rule is to be followed. Overloaded method must change the argument list)

Invoking overloaded methods that take object references rather than primitive type is a little more interesting. Say you have the overloaded method such that one version takes an animal and another version takes a horse (horse extends animal). If you pass a Horse object in the method invocation, you will invoke the overloaded version that takes Horse.

**class Animal**

**{**

**}**

**class Horse extends Animal**

**{**

**}**

**class UseAnimals**

**{**

**public void doStuff(Animal s)**

**{**

**System.out.println("In the animal version");**

**}**

**public void doStuff(Horse h)**

**{**

**System.out.println("In the horse version");**

**}**

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

**{**

**UseAnimals ua=new UseAnimals();**

**Animal animalObj=new Animal();**

**Horse horseObj=new Horse();**

**ua.doStuff(animalObj);**

**ua.doStuff(horseObj);**

**}**

**}**

Now, here, the output is what you expect

In the animal version

In the horse version

**But, what if you use an Animal reference to a horse object?**

**Animal animalRefToHorse=new Horse();**

**ua.doStuff(animalRefToHorse);**

Which of the overloaded versions is invoked? You might want to answer, “The want that takes a Horse, since, ultimately, the Animal class object animalRefToHorse is holding reference of a Horse class, and that is the one which is passed to the method. But that’s not how it works. The preceding code will actually print:

In the animal version.

Even though the actual object at runtime is a Horse and not an Animal. The choice of which overloaded method to call (in other words, the signature of the method) is not dynamically decided at runtime.

**(in case of overloading)**

**Which overridden version of the method to call is decided at runtime based on the object type.**

**Animal animalRefToHorse=new Horse();**

Here, Animal is the reference type.

**new Horse(),** this Horse is the object type.

**class Animal**

**{**

**public void eat()**

**{**

**System.out.println("Generic animal eating generically"):**

**}**

**}**

**public class Horse extends Animal**

**{**

**public void eat()**

**{**

**System.out.println("Horse eating hay");**

**}**

**public void eat(String s)**

**{**

**System.out.println("Horse eating "+s);**

**}**

**}**

Notice that, the Horse class has both overloaded and overridden the eat() method.

|  |  |
| --- | --- |
| **Method Invocation Code** | **Result** |
| Animal a=new Animal();  a.eat(); | Generic animal eating generically |
| Horse h=new Horse(); h.eat(); | Horse eating hay |
| Animal ah=new Horse(); ah.eat(); | Horse eating Hay. Polymorphism works: The actual object type (Horse), not the reference type (Animal), is used to determine which eat() is called |
| Horse he=new Horse();  He.eat(“Apples”); | Horse eating Apples  The overloaded eat(String s) method is invoked |
| Animal a2=new Animal();  A2.eat(“treats”) | Compiler error!. Compiler does not see that the animal class does not have an eat() method that takes a string |
| Animal ah2=new Horse();  ah2.eat(“Carrots”); | Compiler error. Compiler will look only at the reference (compile time) and sees that Animal does not have an eat() method that takes a string. Compiler does not care the type of actual object. |

**Difference Between Overridden and Overloaded Methods:**

|  |  |  |
| --- | --- | --- |
|  | **Overloaded Method** | **Overridden Method** |
| Argument(s) | Must change. | Must not change |
| Return Type | Can change | In most cases cannot change until the return type was an object to some superclass in the original method and object to the subclass in the overridden methods |
| Exceptions | Can change | Can reduce or eliminate. Can not throw new or broader checked exceptions |
| Access | Can change | Must not make more restrictive access |
| Invocation | Reference type determines which overloaded version is selected. Happens at compile time. | Object type (in other words, the type of the actual instance on the heap) determines which method is selected. Happens at runtime. |

**Now, a overridden method can return a object to the subclass of the class whose object is returned in the original function. This property is called Covariant Returns.**

**Casting:**

You have seen how it’s both possible and common to use generic reference variable types to refer to more specific object types. (we could use a superclass reference to refer to subclass object types) It’s the heart of the polymorphism. For example, this line of code second nature by now:

**Animal animal=new Dog();**

But what happens when you want to use that animal reference variable to invoke a method which is specific to Dog. (and that method is not defined as a generic method is Animal)

In the following code, we have got an array of Animals, and whenever we find a Dog in the array, we want to do a special Dog thing.

**class Animal**

**{**

**void makeNoise();**

**}**

**class Dog extends Animal**

**{**

**void makeNoise()**

**{**

**System.out.println("Bark");**

**}**

**void playDead()**

**{**

**System.out.println("Roll over");**

**}**

**}**

**public class CastTest**

**{**

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

**{**

**Animal []a={new Animal(), new Dog(), new Animal()};**

**for(Animal animal: a)**

**{**

**animal.makeNoise();**

**if(animal instanceof Dog)**

**//it will check the specific class Type of an object**

**{**

**animal.playDead();**

**}**

**}**

**}**

**}**

It will necessarily generate compilation error.

However, if we change the following code block:   
  
**if(animal instanceof Dog)**

**//it will check the specific class Type of an object**

**{**

**animal.playDead();**

**}**

To

**if(animal instanceof Dog)**

**//it will check the specific class Type of an object**

**{**

**Dog d=(dog) animal;**

**animal.playDead();**

**}**

It will work fine.

Thee new and improved code block contains a cast, which in this case is sometimes called a downcast, because, we are casting down the inheritance tree to a more specific class.

It’s important to know about that the compiler is forced to trust us when we do a downcast, even when we screw up:

**Class Animal**

**{**

**}**

**Class Dog extends Animal**

**{**

**}**

**Class DogTest**

**{**

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

**{**

**Animal animal=new Animal();**

**Dog d=(Dog)animal;**

**}**

**}**

The code will be compiled but it will fail later.

It will generate the ClassCastException at runtime.

java.lang.ClassCastException

Why cannot we trust the compiler to help use out here? Can’t it see that animal is of type Animal? All the compiler can do is to verify that the two types are in the same inheritance tree. So that depending upon whatever code might have come before the downcast, it’s possible that animal is of type Dog. The compiler must allow things that might possibly works at runtime. However, if compiler knows with certainty that the cast would not possibly work, compilation will fail. The following replacement code block will not compile:

**Animal animal=new Animal();**

**Dog d=(Dog) animal;**

**String a=(string) animal;**

In this case, you will get an error something like this: Inconvertible types.

Since, String and Animal do not belong to the same instance tree.

Now, unlike downcasting, upcasting (casting up the inheritance tree) works implicitly. Because, when you upcast you are implicitly restricting the number of methods you can invoke, as opposed to downcasting, which is implied that later on, you might want to invoke a more specific method.

The implicit typecast is always legal for assigning an object of a subtype to a reference of one of its supertype classes (or interfaces). If Dog implements pet **(even interface and class can be part of same inheritance tree? I understand that superclass and subclass are part of same inheritance tree)** and Pet defined beFriendly(), then a dog can be implicitly typecast to a pet, but the only Dog method you can invoke is beFriendly()

One more thing..If Dog implements Pet, then if Beagle extends Dog, but beagle does not declare that implements Pet, **it is completely Legal and Beagle is still a Pet. Beagle is a pet simply because it extends Dog, and Dog’s already taken care of the pet parts for itself, and for all its children. The Beagle class can always override any method it inherits from Dog, including methods that Dog implemented to fulfill its interface contract.**

**Returning The Value:**

* You can return null in a method which an object reference return type.
* An array is perfectly legal return type.  
  public String[] go()

{

char c=’c’;

return c;

}

* In a method with primitive return type, you can return any value or variable that can be implicitly converted to the declared return type.
* In a method with a primitive return type, you can return any value or variable that can be explicitly cast to the declared return type.  
    
  public int foo()

{

float f=32.5f;

return (int)f;

}

* You must not return anything from a void return type.
* In a method with an object reference return type, you can return any object type that can be implicitly cast to the declared return type.

**Constructors:**

**Concurrency:**

In java, java.util.concurrent package helps us to deal with concurrency.

There are two packages under it. Named java.util.concurrent.atomic and java.util.concurrent.locks.

**Atomic Variables:**

**public class Counter**

**{**

**private int count;**

**public void increment()**

**{**

**count++;**

**//just because, it is a single line does not mean it is thread safe**

**//just because, it is a single line, it does not mean it is atomic**

**}**

**public int getValue()**

**{**

**return count;**

**}**

**}**

Now, the full program is the following:

**class Counter**

**{**

**private int count;**

**//now, we dont mention any constructor. So, default constructor is there. And, since, we did not explicitly override it**

**//it can be invoked**

**public void increment()**

**{**

**count++;**

**//just because, it is a single line does not mean it is thread safe**

**//just because, it is a single line, it does not mean it is atomic**

**}**

**public int getValue()**

**{**

**return count;**

**}**

**}**

**class IncrementerThread extends Thread**

**{**

**private Counter counter;**

**//Now, if all instances passed the same counter reference by value?**

**public IncrementerThread(Counter counter)**

**{**

**this.counter=counter;**

**}**

**public void run()**

**{**

**for(int i=0;i<10000;i++)**

**{**

**//but counter's private value count is not initialized to any value**

**counter.increment();**

**}**

**//so, we can understand that run function is not atomic**

**//but, if it was atomic, what would happen?**

**//it will increment the counter by 10000**

**}**

**}**

**public cl{**

**public static void main(String args[]) throws InterruptedException**

**{**

**Counter counter=new Counter();**

**System.out.println("Getting the initial value of the counter: "+counter.getValue());**

**IncrementerThread it1=new IncrementerThread(counter);**

**IncrementerThread it2=new IncrementerThread(counter);**

**it1.start();**

**//the thread (it1) will start it's execution. So, it's run function will be executed**

**System.out.println("The current value of the counter is: "+counter.getValue());**

**//Now, though thread 1 starts incrementing here, counter's variable remain 0**

**it2.start();**

**System.out.println("The current value of the counter is: "+counter.getValue());**

**//the thread (it2) will start it's execution. So, it's run function will be executed**

**it1.join();**

**System.out.println("The current value of the counter is: "+counter.getValue());**

**//the parent thread (which is main thread) here will wait for thread 1 to finish**

**it2.join();**

**//the parent thread (which is main thread) here will wait for thread 2 to finish**

**System.out.println("the current value of the counter is: "+counter.getValue());**

**//at this point it will be 20000**

**//though, according to book it will be rarely 20000**

**//lowest 11972 (how many times did they test?)**

**}**

**}**

Now, this can generate some pretty unexpected result.

Why, because, if you think count++ is an atomic operation, it is a trap.

Because, a set of thing will happen.

Suppose, at processor level it is the INC count instruction.

Now, First, it (will it be checked by operating system. Because, operating system itself is a program) will be checked whether count is present in the page table or not. (page table is necessarily an index mechanism for all memory units present in secondary disk storage and Cache and some part of RAM. If from page table it is known that the count variable is not present in the main memory or RAM, it is needed to be loaded from secondary disk storage to main memory or RAM. Now, it is to be checked that if any page is free. Otherwise, we need some page scheduling algorithm to free some page slot. After that, it is loaded into page table. Now, if page fault happens (that count is not present in main memory) the instruction (INC) is to be fetched and decoded again. Now, finally count stored in main memory is incremented. Now, after incrementing, it needs to be reflected back in the secondary disc storage. Now, that is depended on something else. Now, the steps I describes here, might be wrong. However, that much task is required for incrementing a simple variable.

So, the simple statement count++ will be translated by java compiler into multiple java bytecode instruction. Now, JIT (just in time Compiler) based nature of most java runtime environments means you don’t know when or if the count++ statement will be translated into native CPU instructions and whether it ends up as a single instruction or several. However, we should always act as if a single line of java code takes multiple steps to complete. (Because, whether an erroneous result will be obtained is depended on many factors: The JIT (just In Time Compiler) nature of the compiler, the type of CPU(Do both threads in the example run concurrently or in sequence? Here, a large loop count was used in order to make the threads run longer and be more likely to execute concurrently instead of executing in sequence)

Now, java.util.concurrent.atomic gives you a mechanism.

**class Counter**

**{**

**private AtomicInteger count;**

**//now, we dont mention any constructor. So, default constructor is there. And, since, we did not explicitly override it**

**//it can be invoked**

**public void increment()**

**{**

**count.getAndIncrement();**

**//just because, it is a single line does not mean it is thread safe**

**//just because, it is a single line, it does not mean it is atomic**

**}**

**public int getValue()**

**{**

**return count.intValue;**

**}**

**}**

Here, is an implementation of counter which is thread safe and lock free.

That is why atomic variables comes into scenario.

java.util.concurrent.atomic defines many atomic Classes.

**AtomicBoolean**

A boolean value that may be updated atomically.

**AtomicInteger**

An int value that may be updated atomically.

**AtomicIntegerArray**

An int array in which elements may be updated atomically.

**AtomicIntegerFieldUpdater<T>**

A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.

**AtomicLong**

A long value that may be updated atomically.

**AtomicLongArray**

A long array in which elements may be updated atomically.

**AtomicLongFieldUpdater<T>**

A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.

**AtomicMarkableReference<V>**

An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.

**AtomicReference<V>**

An object reference that may be updated atomically.

**AtomicReferenceArray<E>**

An array of object references in which elements may be updated atomically.

**AtomicReferenceFieldUpdater<T,V>**

A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.

**AtomicStampedReference<V>**

An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically.

**DoubleAccumulator**

One or more variables that together maintain a running double value updated using a supplied function.

**DoubleAdder**

One or more variables that together maintain an initially zero double sum.

**LongAccumulator**

One or more variables that together maintain a running long value updated using a supplied function.

**LongAdder**

One or more variables that together maintain an initially zero long sum.

Java.util.concurrent.Atomic is a small toolkit of classes that support lock-free thread-safe programming on single variables.

Now, go back to the previous example. After that, we will discuss the other functionality this java.util.concurrent.Atomic provides.

**class Counter**

**{**

**private AtomicInteger count;**

**//now, we dont mention any constructor. So, default constructor is there. And, since, we did not explicitly override it**

**//it can be invoked**

**public void increment()**

**{**

**count.getAndIncrement();**

**//just because, it is a single line does not mean it is thread safe**

**//just because, it is a single line, it does not mean it is atomic**

**}**

**public int getValue()**

**{**

**return count.intValue;**

**}**

**}**

Now, in reality, even a method such as getAndIncrement() still takes several steps to execute. The reason this implementation is thread safe is something called CAS (Compare and Swap)

**Some common functions this atomic package offers:**

**addAndGet(int)**

Atomically adds the given value to the current value.

**getAndAdd(int)**

Now, in both case, I guess the order is different.

**compareAndSet(int expect, int update)**

Atomically sets the value to the given updated value if the current value == the expected value.

**getAndDecrement()**

**getAndIncrement()**

**getAndSet(int newValue)**

Atomically sets to the given value and returns the old value.

**Locks:**

Atomic can provide thread safety (without lock) for a single variable. For two or more than two variables, we need lock.

The things java.util.concurrent.locks provide:

The ability to duplicate traditional synchronized blocks.

Nonblock scoped blocking:-obtain a lock in one method and release it in another .

Multiple wait/notify/notifyAll pools per lock: thread can select which pool (condition) they wait on.

The ability to attempt to acquire a lock and take an alternative action if locking fails

An implementation of multiple -reader, single writer block. (Now, this is ReentrantReadWriteLock)

Now, before we proceed into details of java.util.concurrent.locks we will first revisit few concepts:

**ReentrantLock:**The java.util.concurrent.locks.lock interface provides the outline of the new form of locking provided by java.util.Concurrent.locks package.

Now,

**Java.util.concurrent.locks.ReentrantLock** class provides the implementation of this interface.

**Now, a basic example of synchronized block vs Reentrant lock block:**

**Object obj=new Object();**

**synchronized(obj)**

**{**

**//traditional locking, blocks until acquired  
 //work**

**//release lock automatically**

**}**

Here is an equivalent piece of code using java.util.concurrent.locks package. Notice how ReentrantLock can be stored in a lock reference because it implements the lock interface. (this is another feature provided by polymorphism)

**Lock lock=new ReentrantLock();**

**Lock.lock();**

**try**

**{**

**//do work here**

**}**

**finally**

**{**

**Lock.unlock();**

**//to ensure that lock is released**

**//must manually release**

**}**

**It is recommended that you always follow the lock() method with try-finally block which releases the lock.**

A slight modification:

**Lock lock=new ReentrantLock();**

**Boolean locked=lock.tryLock();**

**If(locked)**

**{**

**try**

**{**

**//do work here**

**}**

**finally**

**{**

**Lock.unlock();**

**//to ensure that lock is released**

**//must manually release**

**}**

**}**

Now, as you can see, with the traditional locking mechanism, it also provides the ability to attempt to acquire a lock and take an alternative action if locking fails.

Now, there is also a variation of the trylock method that allows you to specify an amount of time you are willing to wait to acquire the lock:

**Lock lock=new ReentrantLock();**

**try**

**{**

**Boolean locked=lock.tryLock(3,TimeUnit.SECONDS);**

**If(locked)**

**{**

**try**

**{**

**}**

**Finally**

**{**

**lock.unlock();**

**}**

**}**

**}**

**catch(InterruptedException ex)**

**{**

**}**

Now, the function lock() represents traditional locking mechanism. The tryLock(0 function is more rich.

The tryLock function can offer you deadlock avoidance. With traditional synchronization, you must acquire locks in the same order across all the threads. For example, if you have two objects to lock against:

Object o1=new Object();

Object o2=new Object();

And you synchronize using the internal lock flags of both objects:

**synchronized(o1)**

**{**

**//thread A could pause here**

**synchronized(o2)**

**{**

**//work**

**}**

**}**

Now, you cannot acquire the lock in opposite order in another function because, it would lead to deadlock.

Now, look at similar example using a ReentrantLock(). start by creating two locks:

Lock l1=new ReentrantLock();  
Lock l2=new ReentrantLock();

Next, you acquire both locks in thread A:

**boolean aq1=l1.tryLock();**

**boolean aq2=l2.tryLock();**

**try**

**{**

**If(aq1&&aq2)**

**{**

**//work**

**}**

**}**

**finally**

**{**

**If(aq2) l2.unlock();**

**If(aq1) l1.unlock();**

dont unlock if not locked

**}**

Notice the example is careful to a;ways unlock any acquired block. But only the lock(s) that were acquired.

A ReentrantLock has an internal counter that keeps track of the number of times it has been locked/unlocked. **And it is an error** to unlock without a corresponding successful lock operation. It a thread tries to release a lock that is does not own, an IllegalMonitorStateException will be thrown.

Now, suppose, the old ordering is in **Thread A**.

**boolean aq1=l1.tryLock();**

**boolean aq2=l2.tryLock();**

**try**

**{**

**If(aq1&&aq2)**

**{**

**//work**

**}**

**}**

**finally**

**{**

**If(aq2) l2.unlock();**

**If(aq1) l1.unlock();**

**Don’t unlock if not locked**

**}**

Now, in **thread B,** suppose, the following happened:

**boolean aq2=l2.tryLock();**

**boolean aq1=l1.tryLock();**

**try**

**{**

**If(aq1&&aq2)**

**{**

**//work**

**}**

**finally**

**{**

**If(aq1) l1.unlock();**

**If(aq2) l2.unlock();**

**}**

**}**

Now, even if the thread A was only in possession of l1 lock, there is no possibility that thread B could block, because we use the non blocking trylock method. Using this technique, you can avoid the deadlocking scenarios. But, you must deal with the possibility that both locks could not be acquired. Using a simple loop, you can repeatedly attempt to obtain both locks until successful.

However, this approach will generate the spinlock situation (this approach is CPU intensive)

**Loop: while(true)**

**{**

**boolean aq2=l2.tryLock();**

**boolean aq1=l1.tryLock();**

**try**

**{**

**If(aq1 && aq2)**

**{**

**Break Loop;**

**}**

**}**

**finally()**

**{**

**If(aq2) l2.unlock();**

**If(aq1) l1.unlock();**

**}**

**}**

Now, as we already mentioned, this will generate spinlock situation and waste CPU cycle as a resource. One way to avoid spinlock situation in this scenario is by introducing a short random delay with Thread.sleep(int) any time you fail to acquire both locks.

Now, there are better ways than this naive solution.

You can use wait, notify, notifyAll method.

**Condition:**

A condition provides the equivalent of traditional wait, notify and notifyall methods. The traditional wait and notify mechanism allow developers to implement an await/signal mechanism. You use an await/pattern when you would use locking, but with the added stipulation(means condition or agreement) of trying to avoid spinning.

Now, **java.util.concurrent.locks.Condition** interface is the modern replacement for wait)() and notify(). A three part code example shows you how to use a condition.

**Part 1:**

**Lock lock=new ReentrantLock();**

**Condition blockingPoolA=lock.newCondition();**

Now, as you can see a Condition (yes, an interface base reference can be created) is created from a Lock object.

Now, when your thread reaches a point where it must delay until another thread performs an activity, you **“await”** the completion of that other activity. Before calling await, you must have locked the **Lock** used to produce the Condition. It is possible that the awaiting thread may be interrupted and you must handle the possible InterruptedException. When you **call the await method, the lock associated with the condition is released.** However, before the await method returns, the lock will be reacquired. In order to use a Condition, a thread must first acquire a lock.

**Part 2:**

**lock.lock(); //this lock is is used to created the condition**

**try**

**{**

**blockingPoolA.await();**

//blockingPoolA is the Condition object

**}**

**catch(InterruptedException e)**

**{**

//interrupted during await

**}**

**finally**

**{**

**lock.unlock();**

//to ensure that lock must release manually  
**}**

In another thread, you perform the activity that the first thread was waiting one and then signal that first thread to resume. (return from the wait method)

**Part three of the Condition example is run is a different thread than part two. This part causes the thread waiting in the part two to wake up:**

**lock.lock();**

**try**

**{**

**blockingPoolA.signalAll();**

//wake all awaiting thread

**}**

**finally**

**{**

**Lock.unlock();**

//now an awaking thread can run

**}**

Now, the signalAll() method causes all threads awaiting on the same condition to wake up. You can also use the signal () method method to wake up a single awaiting thread.

**Remember that, waking up is not the same thing as proceeding. Each awoken thread will have to reacquire the lock before continuing.**

**import java.util.concurrent.locks.Condition;**

**import java.util.concurrent.locks.Lock;**

**import java.util.concurrent.locks.ReentrantLock;**

**public class SharedFiFoQueue**

**{**

**private Object[] elems = null;**

**private int current = 0;**

**private int placeIndex = 0;**

**private int removeIndex = 0;**

**private final Lock lock = new ReentrantLock();**

**private final Condition isEmpty = lock.newCondition();**

**private final Condition isFull = lock.newCondition();**

**public SharedFiFoQueue(int capacity)**

**{**

**this.elems = new Object[capacity];**

**}**

**public void add(Object elem) throws InterruptedException**

**{**

**lock.lock();**

**while(current >= elems.length)**

**isFull.await();**

**elems[placeIndex] = elem;**

**//We need the modulo, in order to avoid going out of bounds.**

**placeIndex = (placeIndex + 1) % elems.length;**

**++current;**

**//Notify the consumer that there is data available.**

**isEmpty.signal();**

**lock.unlock();**

**}**

**public Object remove() throws InterruptedException**

**{**

**Object elem = null;**

**lock.lock();**

**while(current <= 0)**

**isEmpty.await();**

**elem = elems[removeIndex];**

**//We need the modulo, in order to avoid going out of bounds.**

**removeIndex = (removeIndex + 1) % elems.length;**

**//Notify the producer that there is space available.**

**isFull.signal();**

**lock.unlock();**

**return elem;**

**}**

**}**

It is an implementation of SharedQueue

**Major advantage over traditional wait/notify:**

**Multiple condition can exist for each lock.**

**ReentrantReadWriteLock:**two locks are offered. It can be got using writeLock() and readLock().

**The concept of Reentrancy:**

In computing, a computer program or subroutine is called reentrant if it can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocations complete execution.

**To understand deeply, you have to understand the difference between synchronized bock and ReentrantLock:**

Synchronized block offers an intrinsic lock.

An intrinsic locking mechanism can have some functional limitations, such as:

1.) It is not possible to interrupt a thread waiting to acquire a lock (lock Interruptibly).

2.) It is not possible to attempt to acquire a lock without being willing to wait for it forever (try lock).

3.) Cannot implement non-block-structured locking disciplines, as intrinsic locks must be released in the same block in which they are acquired.

**Fairness:**The ReentrantLock constructor offers a choice of two fairness options: create a non-fair lock or a fair lock. With fair locking, threads can acquire locks only in the order in which they were requested, whereas an unfair lock allows a lock to acquire it out of its turn. This is called barging (breaking the queue and acquiring the lock when it became available).

Fair locking has a significant performance cost because of the overhead of suspending and resuming threads. There could be cases where there is a significant delay between when a suspended thread is resumed and when it actually runs. Let's see a situation:

A -> holds a lock.

B -> has requested and is in a suspended state waiting for A to release the lock.

C -> requests the lock at the same time that A releases the lock, and has not yet gone to a suspended state.

As C has not yet gone to a suspended state, there is a chance that it can acquire the lock released by A, use it, and release it before B even finishes waking up. So, in this context, unfair lock has a significant performance advantage.

**Java.util.Concurrent Collection:**

Without concurrent collections, what will be the problem:

You can understand. No need to give an example.

**CopyOnWrite Collection:**

The **copyOnWrite** collection implements one of the several mechanisms to make a collection thread safe. By using the CopyOnWrite collections.

With copy-on-write collections, you eliminate the need to implement synchronization or locking.

It never modify the internal array of data. Any mutating operation on the list (add, set, remove etc) will create a new copy on the array to be created, which will replace the original read only array. The read only nature of the underlying array in a CopyOnWriteArrayList allows it to be safely shared with multiple threads. Remember that, read only objects (immutable) are always thread safe.

The essential thing to remember with a copy on write collection, is that a thread that is looping through the elements in a collection must keep a reference to the same unchanged elements throughout the duration of the loop, this is achieved with the use of an iterator. Basically, you want to keep using the old, unchanging collection that you began a loop with.

**Difference Between CopyOnWrite collection and Concurrent Collection:**

We already discussed it.

**Concurrent Collection:**

* **ConcurrentHashMap**
* **ConcurrentLinkedDeque**
* **ConcurrentLinkedQueue**
* **ConcurrentSkipListMap**
* **ConcurrentSkipListSet**

Iterator for a concurrent collection is weakly consistent (unlike CopyOnWrite structures). It can return elements from the point in time the iterator was created or later. This means while you are looping through a concurrent collection, you might observe elements that another thread is inserting with methods such as addAll when when concurrently reading from the collection. Similarly, the size method may produce inaccurate results. Imagine attempting to count the number of people in a checkout line at a grocery store. While you are counting the people in the line, some people may join the line and others may leave. Your count might end up close not exact by the time you reach the end. This is the type of behaviour you might see with a weekly consistent collection. The benefit to this type of behaviour is that it is permissible for multiple threads to concurrently read and write a collection without having to create internal copies of the collection (in cache or in RAM). if your application cannot deal with these inconsistencies, you might have to use a copy on write collection.

The ConcurrentHashMap and ConcurrentSkipListMap classes implement the concurrentMap interface. A ConcurrentMap interface enhances a map by adding the atomic **putIfAbsent, remove and replace methods. (simultaneous writes on the same object**  **is not allowed, simultaneous read and write on the same object are not allowed, simultaneous read is allowed**)

**Difference Between HashMap and ConcurrentHashMap:**

* HashMap is non-Synchronized in nature i.e. HashMap is not Thread-safe whereas ConcurrentHashMap is Thread-safe in nature.
* HashMap performance is relatively high because it is non-synchronized in nature and any number of threads can perform simultaneously. But ConcurrentHashMap performance is low sometimes because sometimes Threads are required to wait on ConcurrentHashMap.
* While one thread is Iterating the HashMap object, if other thread try to add/modify the contents of Object then we will get Run-time exception saying ConcurrentModificationException. Whereas In ConcurrentHashMap we wont get any exception while performing any modification at the time of Iteration.

**Default Allowed Concurrency In Case Of ConcurrentHashMap:**

The allowed concurrency among update operations is guided by the optional concurrencyLevel constructor argument (default 16), which is used as a hint for internal sizing. The table is internally partitioned to try to permit the indicated number of concurrent updates without contention. Because placement in hash tables is essentially random, the actual concurrency will vary. Ideally, you should choose a value to accommodate as many threads as will ever concurrently modify the table. Using a significantly higher value than you need can waste space and time, and a significantly lower value can lead to thread contention. But overestimates and underestimates within an order of magnitude do not usually have much noticeable impact. A value of one is appropriate when it is known that only one thread will modify and all others will only read. Also, resizing this or any other kind of hash table is a relatively slow operation, so, when possible, it is a good idea to provide estimates of expected table sizes in constructors.

**Important Constructors Of ConcurrentHashMap:**

**ConcurrentHashMap(int initialCapacity)**

Creates a new, empty map with the specified initial capacity, and with default load factor (0.75) and concurrencyLevel (16).

**ConcurrentHashMap(int initialCapacity, float loadFactor)**

Creates a new, empty map with the specified initial capacity and load factor and with the default concurrencyLevel (16).

**HashMap’s Bucket Size(Load Factor) and Capacity:**An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

A load factor=1 hashmap with number of entries=capacity will statistically have significant amount of collisions (=when multiple keys are producing the same hash). When collision occurs the lookup time increases, as in one bucket there will be >1 matching entries, for which the key must be individually checked for equality.

**HashMap<Integer,String> test = new HashMap<Integer,String>(16,2.0f);**

So, if you initialize a HashMap without specifying an initial size and a load factor it will get initialized with a size of 16 and a load factor of 0.75. This means, once the HashMap is at least (initial size \* load factor) big, so 12 elements big, it will be rehashed, which means, it will grow to about twice the size and all elements will be added anew.

You now set the load factor to 2, which means, now the Map will only get rehashed, when it is filled with at least 32 elements.

What happens now is that elements with the same hash mod bucketcount will be put into the same bucket. Each bucket with more then one element contains a list, where all the elements are put into. Now when you try to lookup one of the elements it first finds the bucket using the hash. Then it has to iterate over the whole list in that bucket to find the hash with the exact match. This is quite costly.

So in the end there is a trade-off: rehashing is pretty expensive, so you should try to avoid it. On the other hand, if you have multiple elements in a bucket, the lookup gets pretty expensive, so you should really try to avoid that as well. So you need a balance between those two. One other way to go is to set the initial size quite high,

but that takes up more memory that is not used.

initial capacity of hashmap \* Load factor of hashmap = 16 \* 0.75 = 12. This represents that until 12th key-value pair hashmap will keep its size to 16 and as soon as 13th item(key-value pair) will come into the Hashmap, it will increase its size from default 24 = 16 buckets to 25 = 32 buckets.

**The functions ConcurrentHashMap offers:**

void clear()

**boolean contains(Object value)**

Legacy method testing if some key maps into the specified value in this table.

This is search by value.

**boolean containsKey(Object key)**

Tests if the specified object is a key in this table.

**boolean containsValue(Object value)**

get, put, remove, remove(Object key, Object value),

**Synchronized block in java:**

Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

(So, different synchronized block on the same object produces a version of locking. (the concept of critical section and remainder section) . The synchronized block on an object creates a critical section for that object. So, it can only have one thread executing inside them at the same time.

All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

**The synchronized keyword can be used to mark four different types of blocks:**

Instance methods

Static methods

Code blocks inside instance methods

Code blocks inside static methods

These blocks are synchronized on different objects. Which type of synchronized block you need depends on the concrete situation.

**Synchronized Instance Methods**

Here is a synchronized instance method:

**public synchronized void add(int value)**

**{**

**this.count += value;**

**}**

Notice the use of the synchronized keyword in the method declaration. This tells Java that the method is synchronized.

A synchronized instance method in Java is synchronized on the instance (object) owning the method. Thus, each instance has its synchronized methods synchronized on a different object: the owning instance. Only one thread can execute inside a synchronized instance method. If more than one instance exist, then one thread at a time can execute inside a synchronized instance method per instance. One thread per instance.

**Synchronized Static Methods**

Static methods are marked as synchronized just like instance methods using the synchronized keyword. Here is a Java synchronized static method example:

**public static synchronized void add(int value)**

**{**

**count += value;**

**}**

Also here the synchronized keyword tells Java that the method is synchronized.

Synchronized static methods are synchronized on the class object of the class the synchronized static method belongs to. Since only one class object exists in the Java VM per class, only one thread can execute inside a static synchronized method in the same class.

If the static synchronized methods are located in different classes, then one thread can execute inside the static synchronized methods of each class. One thread per class regardless of which static synchronized method it calls.

**Synchronized Blocks in Instance Methods**

You do not have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

Here is a synchronized block of Java code inside an unsynchronized Java method:

**public void add(int value)**

**{**

**synchronized(this)**

**{**

**this.count += value;**

**}**

**}**

This example uses the Java synchronized block construct to mark a block of code as synchronized. This code will now execute as if it was a synchronized method.

Notice how the Java synchronized block construct takes an object in parentheses. In the example "this" is used, which is the instance the add method is called on. The object taken in the parentheses by the synchronized construct is called a monitor object. The code is said to be synchronized on the monitor object. A synchronized instance method uses the object it belongs to as monitor object.

Only one thread can execute inside a Java code block synchronized on the same monitor object.

The following two examples are both synchronized on the instance they are called on. They are therefore equivalent with respect to synchronization:

**public class MyClass**

**{**

**public synchronized void log1(String msg1, String msg2)**

**{**

**log.writeln(msg1);**

**log.writeln(msg2);**

**}**

**public void log2(String msg1, String msg2)**

**{**

**synchronized(this)**

**{**

**log.writeln(msg1);**

**log.writeln(msg2);**

**}**

**}**

**}**

Thus only a single thread can execute inside either of the two synchronized blocks in this example.

Had the second synchronized block been synchronized on a different object than this, then one thread at a time had been able to execute inside each method.

**Synchronized Blocks in Static Methods**

Here are the same two examples as static methods. These methods are synchronized on the class object of the class the methods belong to:

**public class MyClass {**

**public static synchronized void log1(String msg1, String msg2){**

**log.writeln(msg1);**

**log.writeln(msg2);**

**}**

**public static void log2(String msg1, String msg2){**

**synchronized(MyClass.class){**

**log.writeln(msg1);**

**log.writeln(msg2);**

**}**

**}**

**}**

Only one thread can execute inside any of these two methods at the same time.

Had the second synchronized block been synchronized on a different object than MyClass.class, then one thread could execute inside each method at the same time.

Note className.class is required here to synchronize a block (independent on some variables on the class)

Now, in case of synchronized block, all synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

(Now, how this blocking achieved? Is that a spin lock mechanism? Or the thread sleeps for some time and is notified when current thread finishes it’s task. If synchronized block does not contain wait() and in some other place it does not contain notify, will it implement spin lock?

(

here a single CPU is shared among many processes. Busy waiting wastes

CPU cycles that some other process might be able to use productively. This

type of semaphore is also called a because the process "spins" while

waiting for the lock. (Spinlocks do have an advantage in that no context switch

is required when a process must wait on a lock, and a context switch may

take considerable time. Thus, when locks are expected to be held for short

times, spinlocks are useful; they are often employed on multiprocessor systems

where one thread can "spin" on one processor while another thread performs

its critical section on another processor.)

Solution Of Busy Waiting:

**void wait(semaphore \*S)**

**{**

**S->value--;**

**If(value<0)**

**{**

**Add the current process to the S->list**

**Block();**

**}**

**}**

**signal(semaphore \*S)**

**{**

**S->value++;**

**If (s->value <= 0)**

**{**

**remove a process P from S->list;**

**wakeup(P);**

**}**

**}**

)

**wait**

Object wait methods has three variance, one which waits indefinitely for any other thread to call notify or **notifyAll** method on the object to wake up the current thread. Other two variances puts the current thread in wait for specific amount of time before they wake up.

**notify**

notify method wakes up only one thread waiting on the object and that thread starts execution. So if there are multiple threads waiting for an object, this method will wake up only one of them. The choice of the thread to wake depends on the OS implementation of thread management.

**notifyAll**

notifyAll method wakes up all the threads waiting on the object, although which one will process first depends on the OS implementation.

**General Syntax For wait mechanism:**

**wait() :** It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify(). The wait() method releases the lock prior to waiting and reacquires the lock prior to returning from the wait() method. The wait() method is actually tightly integrated with the synchronization lock, using a feature not available directly from the synchronization mechanism. In other words, it is not possible for us to implement the wait() method purely in Java: it is a native method.

General syntax for calling wait() method is like this:

**synchronized( lockObject )**

**{**

**while( ! condition )**

**{**

**lockObject.wait();**

**}**

**//take the action here;**

**}**

**notify() :** It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource. It tells a waiting thread that that thread can wake up. However, the lock is not actually given up until the notifier’s synchronized block has completed. So, if a notifier calls notify() on a resource but the notifier still needs to perform 10 seconds of actions on the resource within its synchronized block, the thread that had been waiting will need to wait at least another additional 10 seconds for the notifier to release the lock on the object, even though notify() had been called.

General syntax for calling notify() method is like this:

**synchronized(lockObject)**

**{**

**//establish\_the\_condition;**

**lockObject.notify();**

**//any additional code if needed**

**}**

**notifyAll() :** It wakes up all the threads that called wait() on the same object. The highest priority thread will run first in most of the situation, though not guaranteed. Other things are same as notify() method above.

General syntax for calling notify() method is like this:

**synchronized(lockObject)**

**{**

**establish\_the\_condition;**

**lockObject.notifyAll();**

**}**

Wait And Notify First Example:

**public class ThreadA**

**{**

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

**{**

**ThreadB b = new ThreadB();**

**b.start();**

//How, can this segment of code is accessed by multiple thread?

**synchronized(b)**

//what’s the point of synchronizing b here?

**{**

**try**

**{**

**System.out.println("Waiting for b to complete...");**

**b.wait();**

//Now, main thread calls b.wait() instead of b.join()

//b.join() makes the current thread(which is main thread) to wait for the thead b

//now, here, we achieve the same thing by b.wait() However, b must notify after it finishes it’s execution

**}**

**catch(InterruptedException e)**

**{**

**e.printStackTrace();**

**}**

**System.out.println("Total is: " + b.total);**

**}**

**}**

**}**

**class ThreadB extends Thread**

**{**

**int total;**

**@Override**

**public void run()**

**{**

**synchronized(this)**

**{**

**for(int i=0; i<100 ; i++)**

**{**

**total += i;**

**}**

**notify();**

**}**

**}**

**}**

However, how the following is also working fine?

**public class ThreadA**

**{**

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

**{**

**ThreadB b = new ThreadB();**

**b.start();**

**synchronized(b)**

**{**

**try**

**{**

**System.out.println("Waiting for b to complete...");**

**b.wait();**

**}**

**catch(InterruptedException e)**

**{**

**e.printStackTrace();**

**}**

**System.out.println("Total is: " + b.total);**

**}**

**}**

**}**

**class ThreadB extends Thread**

**{**

**int total;**

**@Override**

**public void run()**

**{**

**synchronized(this)**

**{**

**for(int i=0; i<100 ; i++)**

**{**

**total += i;**

**}**

**//notify();**

**}**

**}**

**}**

Now, a general producer consumer solution using lock is the following:

**Producer Module:**

**do**

**{**

**produce an item in nextp**

**wait(empty);**

**//Think about it. Empty initially needed to be initialized to n (if there are n slots in the buffer)**

**wait(mutex);**

**//this mutex is relatively simple. This will be either acquired hen an item form the buffer is being consumed or an item is produced**

**add nextp to buffer**

**signal(mutex);**

**signal(full);**

**} while (TRUE);**

**Consumer Module:**

**do**

**{**

**wait (full);**

**//again full mutex will be a counting mutex and must be initialized to 0**

**//now, if 0, that means no item is produced. Hence, it will wait until some element in the buffer is produced and it can consume**

**wait (mutex) ;**

**remove an item from buffer to nextc**

**signal(mutex);**

**signal(empty);**

**consume the item in nextc**

**} while (TRUE);**

The same logic is implemented using wait, notify and synchronized block

**import java.util.Vector;**

**class Producer extends Thread**

**{**

**static final int MAXQUEUE = 5;**

**Queue size**

**private Vector messages = new Vector();**

**@Override**

**Run is an function which must be overrides in any class which overrides Thread class**

**public void run()**

**{**

**try**

**{**

**while (true)**

**{**

**putMessage();**

**Now, consumer’s primary job is to put a message in a buffer**

**//sleep(5000);**

**We will later see the implementation of it**

**}**

**}**

**catch (InterruptedException e)**

**{**

**}**

**}**

**private synchronized void putMessage() throws InterruptedException**

**{**

**Now, it is synchronized function. What does it mean?**

**Means, the variables in which the function operates, those variables will be accessed in a synchronized manner**

**Now, this variable will operate on messages vector. Hence, messages vector will be accessed in a synchronized manner. So, when one thread is doing some operation on it, another thread cannot**

**while (messages.size() == MAXQUEUE)**

**{**

**wait();**

**}**

**messages.addElement(new java.util.Date().toString());**

**System.out.println("put message");**

**notify();**

**//Later, when the necessary event happens, the thread that is running it calls notify() from a block synchronized on the same object.**

**}**

**// Called by Consumer**

**public synchronized String getMessage() throws InterruptedException**

**{**

**notify();**

**while (messages.size() == 0)**

**{**

**wait();//By executing wait() from a synchronized block, a thread gives up its hold on the lock and goes to sleep.**

**}**

**String message = (String) messages.firstElement();**

**messages.removeElement(message);**

**return message;**

**}**

**}**

**class Consumer extends Thread**

**{**

**Producer producer;**

**Consumer(Producer p)**

**{**

**producer = p;**

**}**

**@Override**

**public void run()**

**{**

**try**

**{**

**while (true)**

**{**

**String message = producer.getMessage();**

**System.out.println("Got message: " + message);**

**//sleep(200);**

**}**

**}**

**catch (InterruptedException e)**

**{**

**e.printStackTrace();**

**}**

**}**

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

**{**

**Producer producer = new Producer();**

**producer.start();**

**new Consumer(producer).start();**

**}**

**}**

However, this design is slightly complex . though, I perfectly understand it.

**Daemon thread in Java**

Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection.

They can not prevent the JVM from exiting when all the user threads finish their execution.

JVM terminates itself when all user threads finish their execution (JVM is invoked by java command)

If JVM finds running daemon thread, it terminates the thread and after that shutdown itself. JVM does not care whether Daemon thread is running or not.

It is an utmost low priority thread.

**Functions:**

**void setDaemon(boolean status):** This method is used to mark the current thread as daemon thread or user thread. For example if I have a user thread tU then tU.setDaemon(true) would make it Daemon thread. On the other hand if I have a Daemon thread tD then by calling tD.setDaemon(false) would make it user thread.

**Syntax:**

public final void setDaemon(boolean on)

**parameters:**

**on :** if true, marks this thread as a daemon thread.

**exceptions:**

**IllegalThreadStateException:** if only this thread is active.

**SecurityException:** if the current thread cannot modify this thread.

**boolean isDaemon():**

This method is used to check that current is daemon. It returns true if the thread is Daemon else it returns false.

**When does a thread throws IllegalStateException when it is tried to make as daemon?**

**class TestDaemonThread2 extends Thread**

**{**

**public void run()**

**{**

**System.out.println("Name: "+Thread.currentThread().getName());**

**System.out.println("Daemon: "+Thread.currentThread().isDaemon());**

**}**

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

**{**

**TestDaemonThread2 t1=new TestDaemonThread2();**

**TestDaemonThread2 t2=new TestDaemonThread2();**

**t1.start();**

**t1.setDaemon(true);//will throw exception here**

**t2.start();**

**}**

**}**

Here, we cannot set t1 as daemon thread because we call the start method already.

**Application Of Daemon Thread in Java:**

We can have timer thread as daemon thread. Because when all other threads exit, there is no use of timer thread, hence, it can be a daemon thread

Implement Runnable vs Extend Thread in Java:

We can define a thread in the following two ways:

By extending Thread class

By implementing Runnable interface

In the first approach, Our class always extends Thread class. There is no chance of extending any other class. Hence we are missing Inheritance benefits. In the second approach, while implementing Runnable interface we can extends any other class. Hence we are able to use the benefits of Inheritance.

Because of the above reasons, implementing Runnable interface approach is recommended than extending Thread class.

**The significant differences between extending Thread class and implementing Runnable interface:**

* When we extend Thread class, we can’t extend any other class even we require and When we implement Runnable, we can save a space for our class to extend any other class in future or now.
* When we extend Thread class, each of our thread creates unique object and associate with it. When we implements Runnable, it shares the same object to multiple threads.
* Extending Thread class introduces tight coupling as the class contains code of Thread class and also the job assigned to the thread implementing Runnable interface introduces loose coupling as the code of Thread is separate form the job of Threads.

**An example of Runnable Interface Implementing:**

**class ImplementsRunnable implements Runnable**

**{**

**private int counter = 0;**

**public void run()**

**{**

**counter++;**

**System.out.println("ImplementsRunnable : Counter : " + counter);**

**}**

**}**

**class ExtendsThread extends Thread**

**{**

**private int counter = 0;**

**public void run()**

**{**

**counter++;**

**System.out.println("ExtendsThread : Counter : " + counter);**

**}**

**}**

**public class ThreadVsRunnable**

**{**

**public static void main(String args[]) throws Exception**

**{**

**//Multiple threads share the same object.**

**ImplementsRunnable rc = new ImplementsRunnable();**

**Thread t1 = new Thread(rc);**

**t1.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**Thread t2 = new Thread(rc);**

**t2.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**Thread t3 = new Thread(rc);**

**t3.start();**

**//Creating new instance for every thread access.**

**ExtendsThread tc1 = new ExtendsThread();**

**tc1.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**ExtendsThread tc2 = new ExtendsThread();**

**tc2.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**ExtendsThread tc3 = new ExtendsThread();**

**tc3.start();**

**}**

**}**

**The output is something like this:**

ImplementsRunnable : Counter : 1

ImplementsRunnable : Counter : 2

ImplementsRunnable : Counter : 3

ExtendsThread : Counter : 1

ExtendsThread : Counter : 1

ExtendsThread : Counter : 1

It proves differences given above. I make a slight modification in code given below:

**class ImplementsRunnable implements Runnable**

**{**

**private int counter = 0;**

**public void run()**

**{**

**counter++;**

**System.out.println("ImplementsRunnable : Counter : " + counter);**

**}**

**}**

**public class Runnable**

**{**

**public static void main(String args[]) throws Exception**

**{**

**//Multiple threads share the same object.**

**ImplementsRunnable rc = new ImplementsRunnable();**

**Thread t1 = new Thread(rc);**

**t1.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**Thread t2 = new Thread(rc);**

**t2.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**Thread t3 = new Thread(rc);**

**t3.start();**

**}**

**}**

Now, this is a sample of Runnable interface using. Now, ImplementsRunnable class which implements Runnable, can be shared among various threads.

Now, how to pass that Runnable object in thread? Pass it in constructor during thread creation.

**Important Note about multi-threading:**

**NOTE :** In the case of multithreading we cant predict the exact order of output because it will vary from system to system or JVM to JVM.

**Generics Type:**

Before java 5, there was no syntax for declaring a type-safe collection. So, before java 5, to create an ArrayList of strings, you had to say:

**ArrayList myList=new ArrayList();**

Or the polymorphic equivalent:

**List myList=new ArrayList();**

There was no syntax which let you specify that myList will take strings and only strings. And with no way to specify for the ArrayList, the compiler could not enforce that you put only things of the specified type into the list. (before java 5)

**The Legacy Way To Do Collections:**

Here’s a review of pre-java 5 ArrayList intended to hold strings. (we say **intended** because, that’s about all you had-good intentions -to make sure that the ArrayList would hold only strings)

**List myList=new ArrayList();**

**myList.add(“Fred”);//ok, it will hold strings**

**myList.add(new dog()); //and it will hold dogs,too**

**myList.add(new Integer(42));//and integers**

A non-generic collection can hold any kind of object.

And since a non generic collection could hold anything, the methods that get objects out of the collection could have only one kind of return type. (java.lang.Object)

**(The same idea can actually be used to create a non generic collection in java in modern days. Because, every object of some class can be implicitly typecasted to java.lang.Object class as it is the parent class)**

Or, you can simply do;

**import java.util.ArrayList;**

**class Dog**

**{**

**}**

**public class Test2**

**{**

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

**{**

**ArrayList myList=new ArrayList();**

**myList.add("Fred");**

**myList.add(new Dog());**

**myList.add(new Integer(42));**

**}**

**}**

This will not generate any compilation error. It will just generate some warning.

**Note:** Test2.java uses unchecked or unsafe operations.

**Note:** Recompile with -Xlint:unchecked for details.

**Now, the major problem with non generic class is,** you could not guarantee what was coming out really. For instance, if you create a list of Objects and pushing various type of objects in the list, when you access the data stored in the list, you have to typecast it properly. Otherwise, you will get TypeCastException. It wont generate any compile time error, because, **as long as the original class and the class toy which it is typecasted belongs to the same inheritance tree, there will be no compilation error.**

**Java Generic Class:**

**import java.util.ArrayList;**

**class GenericStack<T>**

**{**

**private ArrayList<T> container;**

**private int top;**

**GenericStack()**

**{**

**container=new ArrayList<>();**

**top=0;**

**}**

**public int size()**

**{**

**return top;**

**}**

**public void push(T item)**

**{**

**container.add(top++,item);**

**//add(int index,E item) is called**

**}**

**public T pop()throws Exception**

**{**

**if(top==0)**

**{**

**throw new Exception("The stack is empty");**

**}**

**return container.remove(--top);**

**}**

**public boolean empty()**

**{**

**return (top==0);**

**}**

**}**

**public class GenericTypeExample**

**{**

**public static void main(String args[])throws Exception**

**try**

**{**

**GenericStack<Integer> container=new GenericStack<>();**

**container.push(10);**

**System.out.println("10 is being pushed in stack");**

**container.push(20);**

**System.out.println("20 is being pushed in stack");**

**container.push(30);**

**System.out.println("30 is being pushed in stack");**

**System.out.println("the element is popped: "+container.pop());**

**System.out.println("the element is popped: "+container.pop());**

**System.out.println("the element is popped: "+container.pop());**

**System.out.println("the element is popped: "+container.pop());**

**}**

**catch(Exception e)**

**{**

**System.out.println("The exception thrown: "+e);**

**}**

**}**

**}**

**Java Generic Interface**

**package java.lang;**

**import java.util.\*;**

**public interface Comparable<T>**

**{**

**public int compareTo(T o);**

**}**

Now, Comparable interface which required in sort function is a great example of generic interface.

**Java Generic Type**

Java Generic Type Naming convention helps us understanding code easily and having a naming convention is one of the best practices of java programming language. So generics also comes with it’s own naming conventions. Usually type parameter names are single, uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are:

E – Element (used extensively by the java collection framework, for example ArrayList, Set etc.)

K – Key (Used in Map)

N – Number

T – Type

V – Value (Used in Map)

**Java Generics Bounded Type Parameters**

Suppose we want to restrict the type of objects that can be used in the parameterized type, for example in a method that compares two objects and we want to make sure that the accepted objects are Comparables. To declare a bounded type parameter, list the type parameter’s name, followed by the extends keyword, followed by its upper bound, similar like below method.

**public static <T extends Comparable<T>> int compare(T t1, T t2)**

**{**

**return t1.compareTo(t2);**

**}**

The invocation of these methods is similar to unbounded method except that if we will try to use any class that is not Comparable, it will throw compile time error.

Bounded type parameters can be used with methods as well as classes and interfaces.

Java Generics supports multiple bounds also, i.e <T extends A & B & C>. In this case A can be an interface or class. If A is class then B and C should be interfaces. We can’t have more than one class in multiple bounds.

Also if there is a class and there are interfaces present as bounds, and if the class bound is not specified first, you will get a compilation error.

For instance, see the following example:

Class A { /\* ... \*/ }

interface B { /\* ... \*/ }

interface C { /\* ... \*/ }

class D <T extends A & B & C> { /\* ... \*/ }

If bound A is not specified first, you get a compile-time error:

class D <T extends B & A & C> { /\* ... \*/ } // compile-time error

**Example:**

public static <T extends Number & Comparable<T>> T maximum(T x, T y, T z)

What does that mean?

The T is a type parameter passed to the generic class Box and should be subtype of Number class and must implements Comparable interface. In case a class is passed as bound, it should be passed first before interface otherwise compile time error will occur.

(that’s why, there are rules. Like, in class D <T extends A & B & C> , only one can be class. And if the class is present, it must be A, neither B nor C)

One good example is:

**public class MaximumTest {**

**// determines the largest of three Comparable objects**

**public static <T extends Comparable<T>> T maximum(T x, T y, T z) {**

**T max = x; // assume x is initially the largest**

**if(y.compareTo(max) > 0) {**

**max = y; // y is the largest so far**

**}**

**if(z.compareTo(max) > 0) {**

**max = z; // z is the largest now**

**}**

**return max; // returns the largest object**

**}**

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

**System.out.printf("Max of %d, %d and %d is %d\n\n",**

**3, 4, 5, maximum( 3, 4, 5 ));**

**System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",**

**6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));**

**System.out.printf("Max of %s, %s and %s is %s\n","pear",**

**"apple", "orange", maximum("pear", "apple", "orange"));**

**}**

**}**

Now,

public static <T extends Comparable<T>> T maximum(T x, T y, T z)

It ensures that x,y and z are comparable objects, right?

**Java Generics and Inheritance:**

We know that Java inheritance allows us to assign a variable A to another variable B if A is subclass of B. So we might think that any generic type of A can be assigned to generic type of B, but it’s not the case. Lets see this with a simple program.

( we can do B b=new A();

//assigning the object of B to the reference of A. Since, A is subclass of B)

**package com.journaldev.generics;**

**public class GenericsInheritance {**

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

**String str = "abc";**

**Object obj = new Object();**

**obj=str; // works because String is-a Object, inheritance in java**

**MyClass<String> myClass1 = new MyClass<String>();**

**MyClass<Object> myClass2 = new MyClass<Object>();**

**//myClass2=myClass1; // compilation error since MyClass<String> is not a MyClass<Object>**

**obj = myClass1; // MyClass<T>’s parent is Object**

**}**

**public static class MyClass<T>{}**

**}**

Now, str can be initialized to obj as both belong to the same inheritance tree and Object class is parent of String class.

**Java Generic Classes and Subtyping**

We can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the extends and implements clauses.

For example, ArrayList<E> implements List<E> that extends Collection<E>, so ArrayList<String> is a subtype of List<String> and List<String> is subtype of Collection<String>.

The subtyping relationship is preserved as long as we don’t change the type argument, below shows an example of multiple type parameters.

**interface MyList<E,T> extends List<E>**

**{**

**}**

(Note that one interface can extends another interface. One interface cannot implement another interface)

**Polymorphism And Generics:**

**List<Integer> myList=new ArrayList<Integer> ();**

This is always supported due to polymorphism.

i.e. we will always be able to assign an ArrayList to a List reference because list is a supertype of ArrayList. (ArrayList implements List)

But, what about this?

**class parent{}**

**class child extends Parent{}**

**List<Parent> myList=new ArrayList<Child>();**

No, it does not work. (will generate a compilation error). There’s a very simple rule here, The type of variable declaration must match the type you pass to the actual object type.

**Polymorphism does not work with same way for generics as it does with arrays.**

So, you are allowed to do this:

**Object[] myArray=new JButton[3];**

But,not this:

**List<Object> myArray=new JButton[3];**

**WildCards In Java Generic Programming:**

Question mark (?) is the wildcard in generics and represent an unknown type. The wildcard can be used as the type of a parameter (parameter to function), field (instance variables?), or local variable and sometimes as a return type.

We can’t use wildcards while invoking a generic method or instantiating a generic class.

**Java Generics Upper Bounded Wildcard:**

Upper bounded wildcards are used to relax the restriction on the type of variable in a method. Suppose we want to write a method that will return the sum of numbers in the list, so our implementation will be something like this.

**public static double sum(List<Number> list)**

**{**

**double sum = 0;**

**for(Number n : list)**

**{**

**sum += n.doubleValue();**

**}**

**return sum;**

**}**

Now, Number is parent class of AtomicInteger, AtomicLong, BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short.

Now the problem with above implementation is that it won’t work with List of Integers or Doubles because we know that List<Integer> and List<Double> are not related, this is when upper bounded wildcard is helpful. We use generics wildcard with extends keyword and the upper bound class or interface that will allow us to pass argument of upper bound or it’s subclasses types.

(now, i don’t understand it. Hence, I will try to understand it by trial and error)

Now, I understand the meaning:

**import java.util.ArrayList;**

**import java.util.List;**

**public class GenericUpperWildCardExample**

**{**

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

**{**

**List<Integer> ints = new ArrayList<>();**

**ints.add(3);**

**ints.add(5);**

**ints.add(10);**

**double sum = sum(ints);**

**System.out.println("Sum of ints="+sum);**

**}**

**public static double sum(List<Number> list){**

**double sum = 0;**

**for(Number n : list){**

**sum += n.doubleValue();**

**}**

**return sum;**

**}**

**}**

Now, Since, java.lang.Integer and java.lang.Double are not related, we cannot pass a List<Integer> to the function since, in the function doubleValue is called.

This will generate compilation time error (yes, not runtime error. Because, function declaration finding (i.e. compiler will check whether the function doubleValue() is declared for Integer class ) is done in compile time. Hence, compilation error)

Now, using upper bound wildcard relax the restriction on the type of variable in a method.

i**mport java.util.ArrayList;**

**import java.util.List;**

**public class GenericsWildcards {**

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

**List<Integer> ints = new ArrayList<>();**

**ints.add(3); ints.add(5); ints.add(10);**

**double sum = sum(ints);**

**System.out.println("Sum of ints="+sum);**

**}**

**public static double sum(List<? extends Number> list){**

**double sum = 0;**

**for(Number n : list){**

**sum += n.doubleValue();**

**}**

**return sum;**

**}**

**}**

Now, using upper bound wildcard we allow to pass argument of upper bound or it’s subclasses types.

Few Misconceptions Regarding This: (and some more trial and errors to understand the actual meaning of upperbound wild card)

**import java.util.ArrayList;**

**import java.util.List;**

**public class Test**

**{**

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

**{**

**List<Integer> ints = new ArrayList<>();**

**ints.add(3); ints.add(5); ints.add(10);**

**double sum = sum(ints);**

**System.out.println("Sum of ints="+sum);**

**}**

**public static double sum(List<Number> list){**

**double sum = 0;**

**for(Number n : list){**

**Double x=(Double)n;**

**sum += x.doubleValue();**

**}**

**return sum;**

**}**

**}**

This is not allowed.

This will give the following compilation error:

double sum = sum(ints);

^

required: List<Number>

found: List<Integer>

reason: actual argument List<Integer> cannot be converted to List<Number> by method invocation conversion

The following is also not allowed:

**import java.util.ArrayList;**

**import java.util.List;**

**public class Test**

**{**

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

**{**

**/\*List<Integer> ints = new ArrayList<>();**

**ints.add(3); ints.add(5); ints.add(10);**

**double sum = sum(ints);**

**System.out.println("Sum of ints="+sum);**

**\*/**

**List<Double> doubles=new ArrayList<>();**

**doubles.add(3.0);**

**doubles.add(5.0);**

**doubles.add(7.0);**

**double sum=sum(doubles);**

**System.out.println("Sum of doubles="+doubles);**

**}**

**public static double sum(List<Number> list){**

**double sum = 0;**

**for(Number n : list){**

**Double x=(Double)n;**

**sum += x.doubleValue();**

**}**

**return sum;**

**}**

**}**

This is also not allowed.

Test.java:18: error: method sum in class Test cannot be applied to given types;

double sum=sum(doubles);

^

required: List<Number>

found: List<Double>

That is why relaxation is allowed. Now, why upperbound? Because, a particular class’s all subclass are being allowed?

**Java Generics Unbounded Wildcard**

Sometimes we have a situation where we want our generic method to be working with all types, in this case unbounded wildcard can be used. Its same as using <? extends Object>.

**public static void printData(List<?> list)**

**{**

**for(Object obj : list)**

**{**

**System.out.print(obj + "::");**

**}**

**}**

We can provide List<String> or List<Integer> or any other type of Object list argument to the printData method. Similar to upper bound list, we are not allowed to add anything to the list.

Now, it is relatively easier.

**Java Generics Lower bounded Wildcard**

Suppose we want to add Integers to a list of integers in a method, we can keep the argument type as List<Integer> but it will be tied up with Integers whereas List<Number> and List<Object> can also hold integers, so we can use lower bound wildcard to achieve this. We use generics wildcard (?) with super keyword and lower bound class to achieve this.

We can pass lower bound or any super type of lower bound as an argument in this case, java compiler allows to add lower bound object types to the list.

**public static void addIntegers(List<? super Integer> list)**

**{**

**list.add(new Integer(50));**

**}**

**So, upper bound wildcard is defined like List<? extends Number>**

**And lower bound wildcard is defined like List<? super Integer>**

A more proper example of this:

**import java.util.ArrayList;**

**import java.util.List;**

**public class GenericsTester {**

**public static void addCat(List<? super Cat> catList) {**

**catList.add(new RedCat());**

**System.out.println("Cat Added");**

**}**

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

**List<Animal> animalList= new ArrayList<Animal>();**

**List<Cat> catList= new ArrayList<Cat>();**

**List<RedCat> redCatList= new ArrayList<RedCat>();**

**List<Dog> dogList= new ArrayList<Dog>();**

**//add list of super class Animal of Cat class**

**addCat(animalList);**

**//add list of Cat class**

**addCat(catList);**

**//compile time error**

**//can not add list of subclass RedCat of Cat class**

**//addCat(redCatList);**

**//compile time error**

**//can not add list of subclass Dog of Superclass Animal of Cat class**

**//addCat.addMethod(dogList);**

**}**

**}**

**class Animal {}**

**class Cat extends Animal {}**

**class RedCat extends Cat {}**

**class Dog extends Animal {}**

**Java Generics Type Erasure:**

Generics in Java was added to provide type-checking at compile time and it has no use at run time, so java compiler uses type erasure feature to remove all the generics type checking code in byte code and insert type-casting if necessary. (generics type checking is removed from bytecode. So that, compatibility is not checked at runtime. Insert type casting if necessary, why? So, it does not generate exception in runtime) Type erasure ensures that no new classes are created for parameterized types (did not understand it); consequently, generics incur no runtime overhead.

public class Test<T extends Comparable<T>>

{

private T data;

private Test<T> next;

public Test(T d, Test<T> n)

{

this.data = d;

this.next = n;

}

public T getData() { return this.data; }

}

Now, T extends Comparable <T> what does that mean?

That means type parameter must support comparison with other instances of its own type, via the Comparable interface.

Now, here, the Java compiler replaces the bounded type parameter T with the first bound interface, Comparable, as below code:

public class Test {

private Comparable data;

private Test next;

public Node(Comparable d, Test n) {

this.data = d;

this.next = n;

}

public Comparable getData() { return data; }

(remember) we can do things like T extends A & B & C

However, If A is class, B and C must be interface in this case.

# Why to Override equals(Object) and hashCode() method ?

HashMap and HashSet use the hashcode value of an object to find out how the object would be stored in the collection, and subsequently hashcode is used to help locate the object in the collection. Hashing retrieval involves:

First, find out the right bucket using hashCode().

Secondly, search the bucket for the right element using equals()

**Case 1: Overriding both equals(Object) and hashCode() method**

You must override hashCode() in every class that overrides equals(). Failure to do so will result in a violation of the general contract for Object.hashCode(), which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable. (-Joshua Bloch)

Here is the contract, from the java.lang.Object specialization:

Whenever it(hashcode) is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. (what is the meaning of this?) This integer need not remain consistent from one execution of an application to another execution of the same application. (Now, what does that mean? According to my deduction, value of hashcode will be same of an object throughout the same execution. But can be very much different in different executions)

If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. (So, in simple language, if two objects are equal, hashcode value for those objects will be same in every execution of the application. Though hashcode value of one object changes during different execution. What does that mean? Suppose, during first execution of the function the object generates a hashcode value 5. During second execution the object generates a hashcode value 7. Now, this can very much happen. Now, suppose , two objects are equal. Then, the second object will also generate the hashcode value 5 during first execution and 7 during second execution)

It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

**// Java program to illustrate**

**// overriding of equals and**

**// hashcode methods**

**import java.io.\*;**

**import java.util.\*;**

**class Geek**

**{**

**String name;**

**int id;**

**Geek(String name, int id)**

**{**

**this.name = name;**

**this.id = id;**

**}**

**@Override**

**public boolean equals(Object obj)**

**{**

**// if both the object references are**

**// referring to the same object.**

**if(this == obj)**

**return true;**

**// it checks if the argument is of the**

**// type Geek by comparing the classes**

**// of the passed argument and this object.**

**// if(!(obj instanceof Geek)) return false; ---> avoid.**

**if(obj == null || obj.getClass()!= this.getClass())**

**return false;**

**// type casting of the argument.**

**Geek geek = (Geek) obj;**

**// comparing the state of argument with**

**// the state of 'this' Object.**

**return (geek.name == this.name && geek.id == this.id);**

**}**

**@Override**

**public int hashCode()**

**{**

**// We are returning the Geek\_id**

**// as a hashcode value.**

**// we can also return some**

**// other calculated value or may**

**// be memory address of the**

**// Object on which it is invoked.**

**// it depends on how you implement**

**// hashCode() method.**

**return this.id;**

**}**

**}**

**// Driver code**

**class GFG**

**{**

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

**{**

**// creating two Objects with**

**// same state**

**Geek g1 = new Geek("aditya", 1);**

**Geek g2 = new Geek("aditya", 1);**

**Map<Geek, String> map = new HashMap<Geek, String>();**

**map.put(g1, "CSE");**

**map.put(g2, "IT");**

**for(Geek geek : map.keySet())**

**{**

**System.out.println(map.get(geek).toString());**

**}**

**}**

**}**

So, public boolean equals()

And public int hashCode() is to be overridden.

Now, the object, which will be a key in the hashmap or hashset, we need to override equals method and hashCode() method.

**Case 2 : Overriding only the equals(Object) method**

If we only override equals(Object) method, when we call map.put(g1, “CSE”); it will hash to some bucket location and when we call map.put(g2, “IT”); it will hash to some other bucket location because of different hashcode value as hashCode() method has not been overridden.

**// Java program to illustrate**

**// Overriding only the equals(Object) method**

**import java.io.\*;**

**import java.util.\*;**

**class Geek**

**{**

**String name;**

**int id;**

**Geek(String name, int id)**

**{**

**this.name = name;**

**this.id = id;**

**}**

**@Override**

**public boolean equals(Object obj)**

**{**

**// if both the object references are**

**// referring to the same object.**

**if(this == obj)**

**return true;**

**// it checks if the argument is of the**

**// type Geek by comparing the classes**

**// of the passed argument and this object.**

**// if(!(obj instanceof Geek)) return false; ---> avoid.**

**if(obj == null || obj.getClass()!= this.getClass())**

**return false;**

**// type casting of the argument.**

**Geek geek = (Geek) obj;**

**// comparing the state of argument with**

**// the state of 'this' Object.**

**return (geek.name == this.name && geek.id == this.id);**

**}**

**}**

**class GFG**

**{**

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

**{**

**// creating two Objects with**

**// same state**

**Geek g1 = new Geek("aditya", 1);**

**Geek g2 = new Geek("aditya", 1);**

**Map<Geek, String> map = new HashMap<Geek, String>();**

**map.put(g1, "CSE");**

**map.put(g2, "IT");**

**for(Geek geek : map.keySet())**

**{**

**System.out.println(map.get(geek).toString());**

**}**

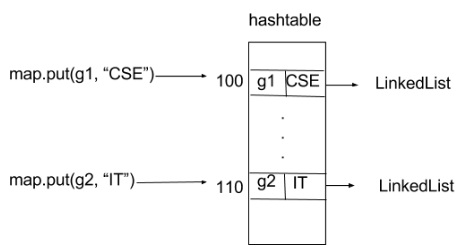
**}**

**}**

**Output:**

CSE

IT



**Case 3: Overriding only hashCode() method**

Consider another example of map :

Map map = new HashMap();

map.put(“xyz”, “CSE”);

map.put(“xyz”, “IT”);

When we call map.put(“xyz”, “CSE”); it will generate hashcode value and stores it to the bucket location that is specified with this address (hashcode value). And when we call map.put(“xyz”, “IT”); it generates same hashcode value as previous entry since key objects are same and hashCode() method has been overridden. So it should replace first with second as per rule. But it didn’t. Reason is, when it iterate through that bucket and seeks to find k such that k.equals(“xyz”) i.e. if searching key already exist. But it fails to find because equals(Object ) method has not been overridden. It is violation of rule of hashing

**// Java program to illustrate**

**// Overriding only hashCode() method**

**import java.io.\*;**

**import java.util.\*;**

**class Geek**

**{**

**String name;**

**int id;**

**Geek(String name, int id)**

**{**

**this.name = name;**

**this.id = id;**

**}**

**@Override**

**public int hashCode()**

**{**

**// We are returning the Geek\_id**

**// as a hashcode value.**

**// we can also return some**

**// other calculated value or may**

**// be memory address of the**

**// Object on which it is invoked.**

**// it depends on how you implement**

**// hashCode() method.**

**return this.id;**

**}**

**}**

**class GFG**

**{**

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

**{**

**// creating two Objects with**

**// same state**

**Geek g1 = new Geek("aditya", 1);**

**Geek g2 = new Geek("aditya", 1);**

**Map<Geek, String> map = new HashMap<Geek, String>();**

**map.put(g1, "CSE");**

**map.put(g2, "IT");**

**for(Geek geek : map.keySet())**

**{**

**System.out.println(map.get(geek).toString());**

**}**

**}**

**}**

**Output:**

CSE

IT

