<http://javarevisited.blogspot.sg/2011/08/enum-in-java-example-tutorial.html>

Java Enum Tutorial: 10 Examples of Enum in Java

**What is Enum in Java**

Enum in Javais a keyword, a feature which is used to represent fixed number of well known values in Java, For example Number of days in Week, Number of planets in Solar system etc. **Enumeration (Enum) in Java** was introduced in JDK 1.5 and it is one of my favorite features of J2SE 5 among Autoboxing and unboxing , Generics, varargs and static import. One of the common use of Enum which emerged in recent years is [Using Enum to write Singleton in Java](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html), which is by far easiest way to implement Singleton and handles several issues related to thread-safety and Serialization automatically. By the way, Java Enum as type is more suitable to represent well known fixed set of things and state,  for example representing state of Order as NEW, PARTIAL FILL, FILL or CLOSED. Enumeration(Enum) was not originally available in Java though it was available in other language like C and C++ but eventually Java realized and introduced Enum on JDK 5 (Tiger) by **keyword Enum**. In this **Java Enum tutorial** we will see different *Enum example in Java* and learn using Enum in Java. Focus of this Java Enum tutorial will be on different features provided by Enum in Java and how to use them. If you have used Enumeration before in C or C++ than you will not be uncomfortable with Java Enum but in my opinion Enum in Java is more rich and versatile than in any other language. . By the way, if you like to learn new concepts using book than you can also see [Java 5.0 Tiger : A Developers notebook](http://www.amazon.com/dp/0596007388/?tag=javamysqlanta-20), I had followed this book while learning Enum, when Java 1.5 was first launched. This book has excellent chapter not only on Enum but also on key features of Java 1.5 and  worth reading.

## How to represent enumerable value without Java enum

Since **Enum in Java** is only available from **Java 1.5** its worth to discuss how we used to represent enumerable values in Java prior JDK 1.5 and without it. I use public static [final constant](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) to replicate enum like behavior. Let’s see an Enum example in Java to understand the concept better. In this example we will use US Currency Coin as enumerable which has values like PENNY (1) NICKLE (5), DIME (10), and QUARTER (25).

**public class** CurrencyDenom {

**public** **static** **final** **int** *PENNY* = 1;

**public** **static** **final** **int** *NICKLE* = 5;

**public** **static** **final** **int** *DIME* = 10;

**public** **static** **final** **int** *QUARTER* = 25;

}

**public class** Currency {

**private int** currency; //CurrencyDenom.PENNY,CurrencyDenom.NICKLE,

CurrencyDenom.DIME,CurrencyDenom.QUARTER

}

 Though this can server our purpose it has some serious limitations:

**1) No Type-Safety**: First of all it’s not [type-safe](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html); you can assign any valid int value to currency e.g. 99 though there is no coin to represent that value.

**2) No Meaningful Printing**: printing value of any of these constant will print its numeric value instead of meaningful name of coin e.g. when you print NICKLE it will print "5" instead of "NICKLE"

**3) No namespace:** to access the currencyDenom constant we need to prefix class name e.g. CurrencyDenom.PENNY instead of just using PENNY though this can also be achieved by using [static import in JDK 1.5](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html)

**Java Enum** is answer of all this limitation. Enum in Java is type-safe, provides meaningful String names and has there own namespace. Now let's see same example using Enum in Java:

**public** **enum** Currency {PENNY, NICKLE, DIME, QUARTER};

Here Currency is our **enum** and PENNY, NICKLE, DIME, QUARTER are **enum constants**. Notice **curly braces around enum constants** because Enum are type like [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) and [interface in Java](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html). Also we have followed similar naming convention for enum like class and interface (first letter in Caps) and since *Enum constants are implicitly static final* we have used all caps to specify them like Constants in Java.

## What is Enum in Java

Now back to primary questions **“What is Enum in java”** simple *answer Enum is a keyword in java* and on more detail term Java Enum is type like class and interface and can be used to define a set of Enum constants. Enum constants are [implicitly static and final](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) and you can not change there value once created. Enum in Java provides type-safety and can be used inside switch statment like int variables. Since enum is a keyword you can not use as variable name and since its only introduced in JDK 1.5 all your previous code which has enum as variable name will not work and needs to be re-factored.

### Benefits of Enums in Java:

1) **Enum is type-safe** you can not assign anything else other than predefined Enum constants to an Enum variable. It is compiler error to assign something else unlike the public static final variables used in Enum int pattern and Enum String pattern.

2) Enum has its own name-space.

3) Best feature of Enum is **you can use Enum in Java inside Switch statement** like int or char primitive data type.we will also see example of using java enum in switch statement in this java enum tutorial.

4) Adding new constants on Enum in Java is easy and you can add new constants without breaking existing code.

## Important points about Enum in Java

1) **Enums in Java are type-safe** and has there own name-space. It means your enum will have a type for example "Currency" in below example and you can not assign any value other than specified in Enum Constants.

**public** **enum** Currency {*PENNY*, *NICKLE*, *DIME*, *QUARTER*};

Currency coin = Currency.PENNY;

coin = 1; //compilation error

2**) Enum in Java are reference type** like [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) or [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html)and you can define constructor, methods and variables inside java Enum which makes it more powerful than Enum in C and C++ as shown in next example of Java Enum type.

3) You can **specify values of enum constants at the creation time** as shown in below example:

**public** **enum** Currency {*PENNY*(1), *NICKLE*(5), *DIME*(10), *QUARTER*(25)};

But for this to work you need to define a member variable and a constructor because PENNY (1) is actually [calling a constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) which accepts int value , see below example.

**public** **enum** Currency {

*PENNY*(1), *NICKLE*(5), *DIME*(10), *QUARTER*(25);

**private** **int** value;

**private** Currency(**int** value) {

**this**.value = value;

        }

};

**Constructor of enum in java** must be [**private**](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html) any other access modifier will result in compilation error. Now to get the value associated with each coin you can define a public getValue() method inside java enum like any normal java class. Also semi colon in the first line is optional.

4) Enum constants are implicitly [static](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html) and [final](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html) and can not be changed once created. For example below code of java enum will result in compilation error:

Currency.PENNY = Currency.DIME;

The final field EnumExamples.Currency.PENNY cannot be re assigned.

5) **Enum in java can be used as an argument on switch statment** and with "case:" like int or char primitive type. This feature of java enum makes them very useful for switch operations. Let’s see an example of how to use java enum inside switch statement:

   Currency usCoin = Currency.DIME;

**switch** (usCoin) {

**case** PENNY:

                    System.out.println("Penny coin");

**break**;

**case** NICKLE:

                    System.out.println("Nickle coin");

**break**;

**case** DIME:

                    System.out.println("Dime coin");

**break**;

**case** QUARTER:

                    System.out.println("Quarter coin");

    }

from JDK 7 onwards you can also [String in Switch case in Java](http://javarevisited.blogspot.sg/2011/08/string-switch-case-jdk7-example.html) code.

6) Since **constants defined inside Enum in Java are final you can safely compare them using "==" equality operator** as shown in following example of  Java Enum:

Currency usCoin = Currency.DIME;

**if**(usCoin == Currency.DIME){

  System.*out*.println("enum in java can be compared using ==");

}

By the way comparing objects using == operator is not recommended, Always use [equals() method](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) or [compareTo() method](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) to compare Objects.

7) Java compiler automatically generates static values() method for every enum in java. Values() method returns array of Enum constants in the same order they have listed in Enum and you can use values() to [iterate](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html) over values of Enum  in Java as shown in below example:

**for**(Currency coin: Currency.values()){

        System.*out*.println("coin: " + coin);

}

And it will print:

**coin: PENNY**

**coin: NICKLE**

**coin: DIME**

**coin: QUARTER**

Notice the order its exactly same **with defined order in enums**.

8) In Java Enum can override methods also. Let’s see an example of overriding toString() method **inside Enum in Java** to provide **meaningful description** for enums constants.

**public** **enum** Currency {

  @Override

**public** String toString() {

**switch** (**this**) {

**case** PENNY:

              System.out.println("Penny: " + value);

**break**;

**case** NICKLE:

              System.out.println("Nickle: " + value);

**break**;

**case** DIME:

              System.out.println("Dime: " + value);

**break**;

**case** QUARTER:

              System.out.println("Quarter: " + value);

        }

**return** **super**.toString();

 }

};

And here is how it looks like when displayed:

Currency usCoin = Currency.*DIME*;

System.out.println(usCoin);

**output:**

**Dime: 10**

9) Two new collection classes **EnumMap and EnumSet** are added into collection package to **support Java Enum**. These classes are high performance implementation of [Map and Set interface in Java](http://javarevisited.blogspot.sg/2012/07/create-read-only-list-map-set-example-java.html) and we should use this whenever there is any opportunity.

10**) You can not create instance of enums by using new operator** in Java because constructor of Enum in Java can only be private and Enums constants can only be created inside Enums itself.

11) Instance of Enum in Java is created when any Enum constants are first called or referenced in code.

12) **Enum in Java can implement the interface** and override any method like normal class It’s also worth noting that Enum in java implicitly implement both [Serializable](http://javarevisited.blogspot.sg/2012/01/serializable-externalizable-in-java.html) and [Comparable](http://javarevisited.blogspot.sg/2011/06/comparator-and-comparable-in-java.html) interface. Let's see and example of **how to implement interface using Java Enum**:

**public** **enum** Currency **implements** Runnable{

  PENNY(1), NICKLE(5), DIME(10), QUARTER(25);

**private** **int** value;

  ............

  @Override

**public** **void** run() {

  System.out.println("Enum in Java implement interfaces");

   }

}

13) **You can define abstract methods inside Enum in Java** and can also provide different implementation for different instances of enum in java.  Let’s see an *example of using* [*abstract method*](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html) *inside enum in java*

**public** **enum** Currency **implements** Runnable{

          PENNY(1) {

                  @Override

**public** String color() {

**return** "copper";

                  }

          }, NICKLE(5) {

                  @Override

**public** String color() {

**return** "bronze";

                  }

          }, DIME(10) {

                  @Override

**public** String color() {

**return** "silver";

                  }

          }, QUARTER(25) {

                  @Override

**public** String color() {

**return** "silver";

                  }

          };

**private** **int** value;

**public** **abstract** String color();

**private** Currency(**int** value) {

**this**.value = value;

          }

          ..............

  }

In this example since every coin will have different color we made the color() method abstract and let each instance of Enum to define   there own color. You can get color of any coin by just calling color() method as shown in below example of java enum:

System.out.println("Color: " + Currency.DIME.color());

**Enum Java valueOf example**

One of my reader pointed out that I have not mention about valueOf method of enum in Java, which is used to convert String to enum in java.  Here is what he has suggested, thanks @ Anonymous

“You could also include **valueOf() method of enum** in java which is added by compiler in any enum along with values() method. **Enum valueOf()** is a static method which takes a string argument and can be used to convert a String into enum. One think though you would like to keep in mind is that valueOf(String) method of enum will throw "**Exception in thread "main" java.lang.IllegalArgumentException: No enum const class**" if you supply any string other than enum values.

Another of my reader suggested about ordinal() and name() utility method of java enum Ordinal method of Java Enum returns position of a Enum constant as they declared in enum while name()of Enum returns the exact string which is used to create that particular Enum constant.” name() method can also be used for [converting Enum to String in Java](http://javarevisited.blogspot.sg/2011/12/convert-enum-string-java-example.html).

That’s all on Java enum , Please share if you have any nice tips on enum in Java  and let us know how you are using java enum in your work. You can also follow some good advice for using Enum by Joshua Bloch in his all time classic book Effective Java. Those advice will give you more idea of using this powerful feature of Java programming language

**Further Reading on Java Enum**

If you like to learn more about this cool feature, I suggest reading following books. Books are one of the best resource to completely understand any topic and I personally follow them as well. Enumeration types chapter from Thinking in Java is particularly useful.

* [Thinking in Java (4th Edition) By Bruce Eckel](http://www.amazon.com/dp/0131872486/?tag=javamysqlanta-20)
* [Effective Java by Joshua Bloch](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20)
* Java 5.0 Tiger: A Developers notebook
* Java 7 Recipes

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html>

Java 1.5 Generics Tutorial: How Generics in Java works with Example of Collections, Best practices, Gotchas

**Java Generics Tutorial**

Generics in Java is one of important feature added in Java 5 along with Enum, autoboxing and varargs , to provide compile time type-safety. Generics is also considered to be one of tough concept to understand in Java and somewhat it’s true as well. I have read many articles on generics in Java, some of them are quite good and detailed but I still felt that those are either too much technical or exhaustively detailed, so I thought to write a simple yet informative article on Java generics to give a head start to beginners without bothering there head too much. In this Java generics tutorial I will cover **How Generics works in Java**, Mysterious **wild-cards in Generics** and some important points about Generic in Java. I will try explaining generics concept in simple words and simple examples. On a different note, If you like to learn new concepts by following books then you should check Java Generics and Collection, one of the best book on Generics, which covers from basics to best practices.

By the way I thought about writing on Java Generics when I completed my post on [Advanced Example of Enum in Java.](http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html) Since Enum and Generics are introduced at same time in JDK 5. If you like to read about generics than you can also check my other tutorials on generics e.g. 10 generics interview question in Java and [Difference between bounded and unbounded wildcards in Generics](http://javarevisited.blogspot.sg/2012/04/what-is-bounded-and-unbounded-wildcards.html).

**What is Generics in Java**

Generic in Java is added to provide compile time type-safety of code and removing risk of ClassCastException at [runtime](http://javarevisited.blogspot.sg/2012/03/what-is-static-and-dynamic-binding-in.html) which was quite frequent error in Java code, for those who doesn’t know what is type-safety at compile time, it’s just a check by compiler that correct Type is used in correct place and there should not be any ClassCastException.  For example HashSet of String will only contain String object and if you try to put Integer or any other object, compiler will complain. Before Java 5 same code will pass compile time check but will fail at runtime which is worse. Generics allows Java programmer to write more robust and **type-safe code**. In my opinion *generics in Java* is much overdue feature given popularity of [Collection framework in java](http://javarevisited.blogspot.com/2011/11/collection-interview-questions-answers.html) and its limitation around handling type-safety. Though Generics may look very complex because of its mysterious angle bracketing <> and various wild cards on Java generics, but once you understand the purpose of Generics in Java or Why Generics is introduced in Java you will be very comfortable and love writing code using Generics. If you are beginner and not familiar with Generics I strongly suggest you put some time and understand the Generics and stop writing collection classes without Generics. It not only saves you from mischievous ClassCastException but also gives you more readability and deterministic behavior from code. In this Java Generics tutorial we will see some important points around Generics and Java and will revise some stuff you may know.

## How Generics works in Java

This is a [popular Java Generics interview question](http://javarevisited.blogspot.sg/2012/06/10-interview-questions-on-java-generics.html) which comes in my mind little late, It didn't come when I first know about generics in Java but a while later, nevertheless I find it quite useful to know about how generics works in java behind the scene. The buzzing keyword is **"Type Erasure"**, you guessed it right it’s the same thing we used to in our schools for erasing our mistakes in writing or drawing :). Same thing is done by Java compiler, when it sees code  written using Generics it completely erases that code and covert it into raw type i.e. code without Generics. All type related information is removed during erasing. So your ArrayList<Gold> becomes plain old [ArrayList](http://javarevisited.blogspot.sg/2011/09/difference-vector-vs-arraylist-in-java.html)  prior to JDK 1.5, formal type parameters e.g. <K, V> or <E> gets replaced by either Object or Super Class of the Type. Also when the translated code is not type correct compiler inserts a type casting operator. This all done behind the scene so you don't need to worry about what important to us is that Java compiler guarantees type-safety and flag any type-safety relate error during compilation. In short Generics in Java is syntactic sugar and doesn’t store any type related information at runtime. All type related information is erased by Type Erasure, this was the main requirement while developing Generics feature in order to reuse all Java code written without Generics.

**Rules and Examples of Generics in Java**

Let’s see some rule of using Generics in Java on Collections, Type safe class and type safe methods with simple examples:

1) Parametrized type like Set<T> is subtype of raw type Set and you can assign Set<T> to Set, following code is legal in Java:

**Set** setOfRawType = **new** [**HashSet**](http://javarevisited.blogspot.com/2012/06/hashset-in-java-10-examples-programs.html)<**String**>();

setOfRawType = **new** **HashSet**<**Integer**>();

2) Set<Object> is setOfAnyType, it can store String, Integer but you can not assign setOfString or setOfInteger to setOfObject using Generics in Java.

**Set**<**Object**> setOfAnyType = **new** **HashSet**<**Object**>();

setOfAnyType.add("abc"); *//legal*

setOfAnyType.add(**new** **Float**(3.0f)); *//legal - <Object> can accept any type*

3)[Set](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html)<?> represents SetOfUnknownType and you can assign SetOfString or SetOfInteger to Set<?> as shown in below example of Generics :

**Set**<?> setOfUnknownType = **new** **LinkedHashSet**<**String**>();

setOfUnknownType = **new** **LinkedHashSet**<**Integer**>();

4)Parametrized Type also follow Inheritance at main Type level means both HashSet<String> and LinkedHashSet<String> are sub types of Set<String> and legal by compiler as shown in following Generics example in Java :

**Set**<**String**> setOfString = **new** **HashSet**<**String**>(); *//valid in Generics*

setOfString = **new** **LinkedHashSet**<**String**>(); *// Ok*

But Inheritance on type parameter is not supported means Set<Object> will not accept Set<String> as per following Generics code.

**Set**<**Object**> SetOfObject = **new** **HashSet**<**String**>(); *//compiler error - incompatible type*

5)Set<? extends Number> will store either Number or sub type of Number like Integer, Float. This is an example of bounded wildcards in Generics

**Set**<? **extends** **Number**> setOfAllSubTypeOfNumber = **new** **HashSet**<**Integer**>(); *//legal - Integer extends Number*

setOfAllSubTypeOfNumber = **new** **HashSet**<**Float**>(); *//legal - because Float extends Number*

6)Set<? super TreeMap> is another example of bounded wildcards, which will store instances of [TreeMap](http://javarevisited.blogspot.sg/2011/12/treemap-java-tutorial-example-program.html) or super class of TreeMap. See following Generics example in Java :

**Set**<? **super** **TreeMap**> setOfAllSuperTypeOfTreeMap = **new** **LinkedHashSet**<**TreeMap**>(); *//legal because TreeMap is superType of itself*

setOfAllSuperTypeOfTreeMap = **new** **HashSet**<**SortedMap**>(); *//legal because SorteMap is super class of TreeMap*

setOfAllSuperTypeOfTreeMap = **new** **LinkedHashSet**<**Map**>(); *//legal since Map is super type of TreeMap*

7) You can not use Generics in .class token, parametrized types like List<String> are not allow along with .class literal.

**List**.**class** *//legal*

**List**<**String**>.**class**  *//illegal*

This is the one place where you need to use Raw type instead of parametrized type in Java.

8) If you are writing Generics method than you need to declare type parameters in method signature between method modifiers and return type as shown in below Java Generics example :

**public** **static** <T> T identical(T source){

**return** source;

 }

failing to declare <T> will result in compile time error. to know more read How to write Generics method in Java

**Generics notations and naming Convention**

One of the reason Generics looks tough is due to non familiarity of various Generics terms and naming conventions. Once you know meaning and name of various terms in generics you will feel more comfortable with Generics in Java. Following are some of the frequently used terms in Generics:

|  |  |
| --- | --- |
| Generic Term | Meaning |
| Set<E> | Generic Type , E is called formal parameter |
| Set<Integer> | Parametrized type , Integer is actual parameter here |
| <T extends Comparable> | Bounded type parameter |
| <T super Comparable> | Bounded type parameter |
| Set<?> | Unbounded wildcard |
| <? extends T> | Bounded wildcard type |
| <? Super T> | Bounded wildcards |
| Set | Raw type |
| <T extends Comparable<T>> | Recursive type bound |

T – used to denote type

E – used to denote element

K – keys

V - values

N – for numbers

**Array and Generics in Java**

1) Arrays doesn't support Generics in Java so you can not create Arrays like T[] which makes gentrifying an existing class hard if you are using arrays. Though there are work around which requires a cast from Object[] to T[] which comes with risk of unchecked cast and warning. For this reason it's better to use Collections classes like ArrayList and HashMap over array. by the way those classes are also implemented on top of array in Java but JDK handles there type-safety by effectively using generics. here is an example of casting Object array to generic array in Java :

/\*\*

 \* Generics and Array doesn't gel very well, Java doesn’t allow Generics array like E[]

 \* @author Javin Paul

 \*/

**public** **class** GenericVsArray {

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

      Holder<**Integer**> numbers = **new** Holder<**Integer**>(10);

      numbers.add(101);

**System**.out.println("Get: " + numbers.get(0));

    }

}

 \* Generic Holder for holding contents of different object type Generic in Java eliminates casting required while calling get(index) from client code

 \* @param <T>

 \*/

**class** Holder<T>{

**private** T[] contents;

**private** **int** index = 0;

**public** Holder(**int** size){

*//contents = new T[size]; //compiler error - generic array creation*

        contents = (T[]) **new** **Object**[size]; *//workaround - casting Object[] to generic Type*

    }

**public** **void** add(T content){

        contents[index] = content;

    }

**public** T get(**int** index){

**return** contents[index];

    }

}

Casting code may generate warning about "unsafe cast" which can be suppressed by using annotation @SuppressWarnings("unchecked") with proper comment that why it will not compromise type-safety. This is also one of the Java Generics best practices suggested in all time classic book Effective Java by Joshua Bloch.

**Generics in Java – Benefits and advantages**

Generics adds lot of value into Java programming language, here are some of important benefits of using Generics in Java:

**Type-safety**

Most important advantage of Generics in Java is **type-safety**. Collections prior to JDK1.5 are not type-safe because they accept Object type argument which allows them to catch all type of objects instead of only required type of object. For example if you want to create an ArrayList of Stocks and you don't want that ArrayList also contain any other asset class you can use generics feature of java to create a **type-safe collection**. Here is an example of using Generics to create a type-safe ArrayList

[**ArrayList**](http://javarevisited.blogspot.sg/2011/05/example-of-arraylist-in-java-tutorial.html)<Stocks> stockList = **new** **ArrayList**<Stocks>();

stockList.add(“coins”); //compiler error , String not allowed

Compiler will guaranteed that only Stock object will be inserted in stockList and will [throw](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html) compiler error if you try to insert different type of Object.

**No Casting**

With Generics you don’t need to cast object , Generics will automatically do that for you. For example here is the code for adding and retrieving an element in List with and without Generics in Java:

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

items.add("chocolates");

**String** item = (**String**) items.get(0)

**List**<**String**> items = **new** **ArrayList**();

items.add("biscuits");

**String** item = items.get(0) *//no cast required*

Since no cast required, result is clear and robust code.

**No ClassCastException**

With Generics compiler ensures that correct types are added into Java collection classes and no cast is required while retrieving element, So there is no risk of ClassCastException at [runtime](http://javarevisited.blogspot.sg/2012/03/what-is-static-and-dynamic-binding-in.html).

## Generics in Java – Important points

Some important feature of Generics in Java worth remembering:

1) One limitation of Generics in Java is that it can not be applied to primitive type, for example you can not create pass primitives in angle bracket that will result in compilation error, for Example ArrayList<int> will result in compilation error, This is little counter-intuitive that [why auto-boxing can not covert int to Integer](http://javarevisited.blogspot.sg/2012/07/auto-boxing-and-unboxing-in-java-be.html). If you try same thing with our Generic Holder class you will get following compilation error:

Holder<**int**> numbers = **new** Holder<**int**>(10); *//compiler error - unexpected type required: reference found:int*

2) Generics in Java eliminates ClassCastException while retrieving objects from Collection, Remember prior to JDK1.5 if you retrieve objects from Collection you first check for a particular type and then cast, no need to do it now.

**ArrayList**<Stocks> stockList = **new** **ArrayList**<StockList>();

Stock sony = **new** Stock("Sony","6758.T");

stockList.add(sony);

Stock retreivedStock = stockList.get(sony); *//no cast requires – automatic casting by compiler*

3) A [parametrized class in Java](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) use formal type parameters to retrieve Type information when instance of parametrized class gets created. In below example of generics class in Java <K,V> are formal parameters.

**interface** Cache <K,V>{

**public** V get();

**public** V put(K key, V value);

}

As per convention followed on Generics version of [Java Collection](http://javarevisited.blogspot.sg/2011/11/collection-interview-questions-answers.html) package we can use <K,V> for key and value type parameters.

4) Generics are often related to **Templates in C++**, though Unlike "Template" in C++, which creates a new type for each specific parametrized type, parametrized class in Java is only compiled once and more importantly there is just one single class file which is used to create instances for all the specific types.

5) Generics in Java can not only apply on [Java Classes](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html) but also on methods, so you can write your own generics methods in Java as shown on Rules of Generics in Java section, here is another example of parametrized method from Java collection package.

boolean add(E o){}

Here E will be replaced by actual type parameter when this method will get called.

6) Another worth noting feature of Generics in Java is its ability to limit Types parameters, for example in parametric declaration of Holder<T extends Closeable>, type parameter list <T extends Closeable> requires that actual parameter T must be either Closeable or sub-type of Closeable. This is called [bounded type parameters in Generics](http://javarevisited.blogspot.sg/2012/04/what-is-bounded-and-unbounded-wildcards.html) . this kind of declaration allows you to call method of Closeable interface without casting type parameter into Closeable. read more about these type parameters in bounded and unbounded wildcards in Generics.

7) **Type inference** : Generics in Java does not support type inference while calling constructor or creating instance of Generic Types until JDK7, In Java 7 along with [Automatic resource management](http://javarevisited.blogspot.sg/2011/09/arm-automatic-resource-management-in.html) and [String in Switch](http://javarevisited.blogspot.sg/2011/08/string-switch-case-jdk7-example.html)  also added a new operator called Diamond operator and denoted by <> which facilitate type inference while creating instance of Generics classes. this helps to reduce redundancy and clutter. here is an example of Diamond operator in Java7 code:

*//prior to JDK 7*

[**HashMap**](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html)<**String**, **Set**<**Integer**>> contacts = **new** **HashMap**<**String**, **Set**<**Integer**>>()

*//JDK 7 diamond operator*

**HashMap**<**String**, **Set**<**Integer**>> contacts = **new** **HashMap**<>()

code with diamond operator is much cleaner than previous one.

On related note Generics in Java supports type inference while calling Generic methods and this feature can be used to create in combination of [Factory design pattern in Java](http://javarevisited.blogspot.sg/2011/12/factory-design-pattern-java-example.html) to create static factory method corresponding to each constructors. for example

*//type inference in generic method*

**public** **static** <K,V> **HashMap**<K,V> newContacts() {

**return** **new** **HashMap**<K,V>();

}

so we can replace call to constructor with this static factory method as shown below :

**HashMap**<**String**, **Set**<**Integer**>> contacts = newContacts();

this can be used as alternative to diamond operator in Java 5 or 6.

## Section for absolute beginners on Generics in Java

If you are absolute beginners in generics those angle bracket "<>" may look strange and unreadable to you. Though is not a complete tutorial on Java Generics and I would suggest you to read Java docs on Generics I will try to give at least some basic idea of generics in Java to get you going. Remember Generics in java are introduced to enforce type-safety especially on collection classes of java which holds type of Object e.g. ArrayList, HashMap.

Type-safety means compiler will verify type of class during compile time and throw compiler error if it found  improper type. For example if an ArrayList of Gold contains Silver compiler will throw error.

ArrayList<Gold> goldList = new ArrayList<Gold>();

<Gold> tells compiler that this ArrayList must contain only Gold.

Generics can also be used to write parametric classes like Cache<Key, Value> on which type of Key and Value can be specified while creating objects.

Parameters used to write code is called **"formal type parameters"** and parameters which passed while creating instance of a generic class in java is called **"actual type parameters".** For example in our generic cache (below) <K, V> are formal parameter while new LRUCache<String, Integer>() will be **actual parameters.**

**Generics wild cards Example in Java**

There are generally two kinds of wild-cards in Generics, Bounded and unbounded. Bounded wildcards can be written in two ways to denote upper bound and lower bound. <?> is called unbounded wildcards because it can accept any Type while <? extends T> and <? super T> are bounded wildcards. To know more about them see my post [Bounded vs Unbounded wildcards in Generics](http://javarevisited.blogspot.sg/2012/04/what-is-bounded-and-unbounded-wildcards.html) . Now let’s see example of different wildcards in Generics:

**<?>**

*"*?" denotes any unknown type, It can represent any Type at in code for. Use this wild card if you are not sure about Type. for example if you want to have a ArrayList which can work with any type than declare it as  "ArrayList<?> unknownList" and it can be assigned to any type of ArrayList as shown in following example of generics in Java:

**ArrayList**<?> unknownList = **new** **ArrayList**<**Number**>();

unknownList = **new** **ArrayList**<**Float**>();

**<? extends T>**

This is little restrictive than previous one it will allow All Types which are either "T" or extends T means subclass of T. for example List<? extends Number> can hold List<Number> or List<Integer>

**ArrayList**<? **extends** **Number**> numberList = **new** **ArrayList**<**Number**>();

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

numberList = **new** **ArrayList**<**Float**>();

**<T super ?>**

This is just opposite of previous one, It will allow T and super classes of T, e.g. List<? super Integer>can hold List<Integer> or List<Number>.

**ArrayList**<? **super** **Integer**> numberList = **new** **ArrayList**<**Number**>();

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

numberList = **new** **ArrayList**<**Float**>(); *//compilation error*

**Generics Best Practices in Java**

After learning about how to use Generics in Java for writing type-safe classes, methods and collection, its worth noting to remember best practices related to Generics coding:

1) Avoid using Raw types for new code. Always use Generics and write parametrized classes and method to get full benefit of compiler checking.

2) Prefer Collection classes over Array in your parametrized class because Generics and Arrays are completely different to each other, Array hold type information at runtime unlike Generics whose type information is erased by type-erasure during run time.

3) Use [Bounded type parameter](http://javarevisited.blogspot.sg/2012/04/what-is-bounded-and-unbounded-wildcards.html) to increase flexibility of method arguments and API

4) Use @SuppressedWarning("unchecked") at as narrow scope as possible like instead of annotating a method, just annotate a line. Also document rational of why this cast is type-safe as [code comments](http://javarevisited.blogspot.sg/2011/08/code-comments-java-best-practices.html).

5) Convert your raw type classes into type-safe parametric class using Generics in Java as and when time allows, that will make code more robust.

Generic in Java is very vast topic and there are lot more to be learn to get expertise on Java Generics. I hope this will serve you a good starting point in terms of reading code written using Generics and get over with complex wild card of Generics. Java Generics is one of the beautiful feature and once you used to it you won’t write classes or methods without generics. Initially Generics looks tough and complex but its worth learning given type-safety benefits it provides. Two things I would suggest you to do as beginner first write collection code always using Generics and write some type-safe classes which can accept parameter e.g. type-safe cache Cache<Key, Value>

**Further Reading on Java Generics**

Generics is a complex topic and effective use of Generics is not easy to learn, but following books has done great job on explaining power of Generics and how to take advantage of that, while writing parameterized interface and classes. This books not only contains good details of Generics and it’s benefit but also best practices of using Generics in Java.

Java Generics and Collection

Effective Java by Joshua Bloch

Java 5.0 Tiger: A Developers notebook

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2012/07/auto-boxing-and-unboxing-in-java-be.html>

What is Autoboxing and Unboxing in Java – Example Tutorial and Corner cases

**What is Autoboxing in Java**

Autoboxing and unboxing is introduced in Java 1.5 to automatically convert primitive type into boxed primitive( Object or Wrapper class). autoboxing allows you to use primitive and object type interchangeably in Java on many places like assignment, method invocation etc. If you have been using Collections like HashMap or ArrayList before Java 1.5 then you are familiar with the issues like you can not directly put primitives into Collections, instead you first need to convert them into Object only then only you can put them into Collections. Wrapper class like Integer, Double and Boolean helps for converting primitive to Object but that clutter the code. With the introduction of **autoboxing and unboxing** in Java this primitive to object conversion happens automatically by Java compiler which makes code more readable. But autoboxing and unboxing comes with certain caveats which needs to be understood before using them in production code and it becomes even more important because they are automatic and can create subtle bugs if you are not sure when autoboxing  in Java code occurs and when unboxing happens. This is my fifth article on features introduced in Java 5 after my post on Java Enum,  How Generics works in Java and varargs example. In this Java tutorial we will see: What is autoboxing and unboxing in Java ?  When autoboxing and unboxing occurs in Java? and things to remember while dealing with primitives and objects in Java with code examples. If you want to understand all the features introduced in Java 5 in much more detail, then I suggest looking at [Core Java Volume 1 9th Edition by Cay S. Horstmann](http://www.amazon.com/Core-Volume-I-Fundamentals-Edition-Series/dp/0137081898?tag=javamysqlanta-20), one of the best core Java book, which covers both concurrency and general features well.

## What is autoboxing and unboxing in Java

When Java automatically converts a primitive type like int into corresponding wrapper class object e.g. [Integer](http://javarevisited.blogspot.sg/2011/08/convert-string-to-integer-to-string.html) than its called **autoboxing**  because primitive is boxed into wrapper class while in opposite case is called **unboxing**, where an Integer object is converted into primitive int. All primitive types e.g. byte, short, char, int, long, float, double and boolean has corresponding wrapper class e.g. Byte, Short, Integer, [Character](http://javarevisited.blogspot.sg/2012/01/get-set-default-character-encoding.html) etc and participate in autoboxing and unboxing. Since whole process happens automatically without writing any code for conversion its called autoboxing and auto unboxing.

**Important point about Autoboxing and Unboxing in Java**

1) Compiler uses valueOf() method to convert primitive to Object and uses intValue(), doubleValue() etc to get primitive value from Object.

2) During autoboxing boolean is converted to Boolean, byte to Byte, [char converted to Character](http://javarevisited.blogspot.sg/2012/02/how-to-convert-char-to-string-in-java.html), float changes to Float, int goes to Integer, long goes to Long and short converts to Short, while in unboxing opposite happens like Float to float.

**When does autoboxing and unboxing occurs in Java**

Autoboxing and unboxing can happen anywhere where an object is expected and primitive type is available for example In method invocation where an object argument is expected,  if you pass primitive, Java automatically converts primitive into equal value Object. Classic use of autoboxing is adding primitive types into Collection like [ArrayList in Java](http://javarevisited.blogspot.sg/2011/06/converting-array-to-arraylist-in-java.html) or creating instance of [parameterized classes](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html) e.g. [ThreadLocal](http://javarevisited.blogspot.sg/2012/05/how-to-use-threadlocal-in-java-benefits.html) which expect Type. here is some code example of auto boxing and unboxing in Java:

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

intList.add(1); *//autoboxing - primitive to object*

intList.add(2); *//autoboxing*

**ThreadLocal**<**Integer**> intLocal = **new** **ThreadLocal**<**Integer**>();

intLocal.set(4); *//autoboxing*

**int** number = intList.get(0); *// unboxing*

**int** local = intLocal.get(); *// unboxing in Java*

You can find all places by applying some common sense as well, just see if an object needed or a primitive type and what is available there but don’t confuse between widening and autoboxing, where former refers to promoting small type into bigger type wherever expected e.g. converting byte to int. I have shared couple of conversion tutorial in java like [String to int conversion](http://javarevisited.blogspot.sg/2011/08/convert-string-to-integer-to-string.html) and  [Double to String conversion](http://javarevisited.blogspot.sg/2011/10/convert-double-to-string-example.html), if you like you also check those.

### Autoboxing and Unboxing Example in Java

In last section we discussed What is autoboxing and unboxing in Java and when does they occur. In short Autoboxing mainly occur in two places one is during assignment and other is during [method invocation](http://javarevisited.blogspot.sg/2012/04/how-to-invoke-method-by-name-in-java.html), let’s see couple of example of autoboxing and unboxing in Java to understand it better :

**Autoboxing and unboxing in assignment:**

This is the most common example of autoboxing in Java, earlier the code was bloated with explicit conversion which is now taken care by compiler.

**//before autoboxing**

**Integer** iObject = Integer.valueOf(3);

Int iPrimitive = iObject.intValue()

**//after java5**

**Integer** iObject = 3; *//autobxing - primitive to wrapper conversion*

**int** iPrimitive = iObject; *//unboxing - object to primitive conversion*

**Autoboxing and unboxing in method invocation:**

This is another place where autoboxing makes your life easy, it allow you to pass Object or primitive interchangeably in a method without explicit conversion:

**public** **static** **Integer** show(**Integer** iParam){

**System**.out.println("autoboxing example - method invocation i: " + iParam);

**return** iParam;

}

*//autoboxing and unboxing in method invocation*

show(3); *//autoboxing*

**int** result = show(3); *//unboxing because return type of method is Integer*

When we call show(Integer) method which accept Integer object with primitive int autoboxing will first convert primitive to object and then call show() method. On the second line unboxing happens because show() method returns Integer while returned value is stored in primitive int variable result.

**Unnecessary Object creation due to Autoboxing in Java**

One of the danger of autoboxing is throw away object which gets created if autoboxing occurs in a loop. Here is an example of how unnecessary object can slow down your application :

**Integer** sum = 0;

**for**(**int** i=1000; i<5000; i++){

   sum+=i;

 }

In this code sum+=i will expand as sum = sum + i and since + operator is not applicable to Integer object it will trigger unboxing of sum Integer object and then autoboxing of result which will be stored in sum which is Integer as shown below :

sum = sum.intValue() + i;

**Integer** sum = **new** **Integer**(result);

here since sum is Integer, it will create around 4000 unnecessary Intger object which are just throw away and if this happens on large scale has It potential to slow down system with [frequent GC](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html) for arithmetic calculation always prefer primitive over boxed primitive and look for **unintentional autoboxing in Java**

**Autoboxing and method overloading in Java**

autoboxing has complicated [method overloading in Java](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html), prior to Java 1.5 value(int) and value(Integer) were completely different and there was no confusion which method will be called based upon type of argument e.g. if you pass int first method will be called and if you pass Integer second method will be called. with autoboxing and unboxing in place it's get trickier. classic example of this is [ArrayList remove() method](http://javarevisited.blogspot.sg/2011/05/example-of-arraylist-in-java-tutorial.html) which is overloaded i.e. remove(index) and remove(Object), Since now ArrayList has two remove() method autoboxing will not occur and respective method will get called as shown in below example of overloading with autoboxing in Java.

**public** **void** test(**int** num){

**System**.out.println("method with primitive argument");

}

**public** **void** test(**Integer** num){

**System**.out.println("method with wrapper argument");

}

*//calling overloaded method*

AutoboxingTest autoTest = **new** AutoboxingTest();

**int** value = 3;

autoTest.test(value); *//no autoboxing*

**Integer** iValue = value;

autoTest.test(iValue); *//no autoboxing*

**Output:**

method with primitive argument

method with wrapper argument

## Things to remember while using autoboxing in Java

So far we have seen *What is autoboxing means in Java* , What is unboxing in Java and when does it occur, But every powerful feature comes with some caveats and corner cases, here are few which is worth remembering while using auto-boxing in Java:

**1) Comparing Objects with equality Operator**

I agree that autoboxing of primitive to Object  adds lot of convenience and reduce verbosity but there are few places where **autoboxing is error prone** e.g. equality operator "==". Since equality operator can be applied on both primitive and Objects it leads to confusion and can cause subtle issues. When you compare two object using "==" operator it compares object's identity and not value and also no auto boxing occur. By the way you its not best practice to use  equality operator to compare Objects, use [equals method](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) instead. here is an example which makes it clear :

**Integer** one = **new** **Integer**(1);

**Integer** anotherOne = **new** **Integer**(1);

if(one == anotherOne){

**System**.out.println("both one are equal");

}else{

**System**.out.println("Both one are not equal");

}

It will print "Both one are not equal" because of no autoboxing. Things gets more confusing when "==" comparison is combined with other logical operators like > and < which *does auto unboxing before comparison*. This one is explained beautifully with an example of Comparator in Effective Java, if you haven't read then go get a copy.

One of my reader Mitchee says that its not clear, so I am updating this section with few more details, Mitchee, let me know if it make sense:

**public** **class** AutoboxingTest {

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

*// Example 1: == comparison pure primitive – no autoboxing*

**int** i1 = 1;

**int** i2 = 1;

**System**.out.println("i1==i2 : " + (i1 == i2)); *// true*

*// Example 2: equality operator mixing object and primitive*

**Integer** num1 = 1; *// autoboxing*

**int** num2 = 1;

**System**.out.println("num1 == num2 : " + (num1 == num2)); *// true*

*// Example 3: special case - arises due to autoboxing in Java*

**Integer** obj1 = 1; *// autoboxing will call Integer.valueOf()*

**Integer** obj2 = 1; *// same call to Integer.valueOf() will return same*

*// cached Object*

**System**.out.println("obj1 == obj2 : " + (obj1 == obj2)); *// true*

*// Example 4: equality operator - pure object comparison*

**Integer** one = **new** **Integer**(1); *// no autoboxing*

**Integer** anotherOne = **new** **Integer**(1);

**System**.out.println("one == anotherOne : " + (one == anotherOne)); *// false*

    }

}

**Output:**

i1==i2 : **true**

num1 == num2 : **true**

obj1 == obj2 : **true**

one == anotherOne : **false**

In first example both argument of == operator is primitive int type so no autoboxing occurs and since 1==1 it prints true

While in second example during assignment to num1, autoboxing occurs which converts primitive 1 into Integer(1) and when we compare num1==num2 unboxing occurs and Integer(1) is converted back to 1 by calling Integer.intValue() method and since 1==1 result is true. In Third example which is a **corner case in autoboxing**, both Integer object are initialized automatically due to autoboxing and since Integer.valueOf() method is used to convert int to Integer and it caches object ranges from -128 to 127, it returns same object both time. In short obj1 and obj2 are pointing to same object and when we [compare two object](http://javarevisited.blogspot.sg/2011/06/comparator-and-comparable-in-java.html) with == operator it returns true without any autoboxing. In last example object are explicitly initialized and compared using equality operator , this time == return false because both one and anotherOne reference variables are pointing to different object.

**2) Mixing object and primitive in equality and relational operator**

Another mistake to avoid while using autoboxing and unboxing in Java is mixing  primitive and Object in equality or relational operator much like [mixing static and non static synchronized method](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html). if we compare one primitive with another object than unboxing of object is occur which could throw NullPointerException if object is null e.g.

**private** **static** **Integer** count;

*//NullPointerException on unboxing*

if( count <= 0){

**System**.out.println("Count is not started yet");

}

**3) Cached Objects**

One more caveat or danger of autoboxing and unboxing is cached object, since valueOf() is used to create boxed primitive and it caches frequently used Object which may behave differently based upon there value as Java only cache integers from -128 to 128.  I have discussed this problem in detail on post [What is wrong while using "==" with auto boxing in Java](http://javarevisited.blogspot.sg/2010/10/what-is-problem-while-using-in.html).

**4) Unnecessary objects and GC overhead**

Last but not least is cost associate on autoboxing and unboxing. Since *autoboxing creates unnecessary object* and if that goes beyond a limit usually outside the range of cached value it can potentially slow your program by frequently causing [garbage collection](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html).

In Summary **autoboxing and unboxing in Java** are great convenience but demands care and awareness while using them. autoboxing and unboxing has several legitimate use case but should not be used with equality operator specially mixing with primitive and object is dangerous. If you like to read books check out [Effective Java](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20) and [Java 5.0 Tiger: A Developer's Notebook](http://www.amazon.com/dp/0596007388/?tag=javamysqlanta-20) , those has some more insightful tips on autoboxing and unboxing in Java.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2011/12/convert-enum-string-java-example.html>

How to parse String to Enum in Java | Convert Enum to String with Example

Converting **Enum into String** and **parsing String to Enum in Java** is becoming a common task with growing use of Enum. Enum is very versatile in Java and preferred choice to represent bounded data and since is almost used everywhere to carry literal value its important to know how to convert Enum to String in Java. In this article we will see both first converting Strings to Enum in Java and than Changing an Enum to String in Java with Example. I thought about this Enum tutorial when I wrote [10 Examples of Enum in Java](http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html). I missed String to Enum conversion and one of reader pointed out that. So here we have now.

## Enum to String to Enum in Java

This article is in continuation of other conversion related post e.g. [how to convert Date to String in Java](http://javarevisited.blogspot.com/2011/09/convert-date-to-string-simpledateformat.html) and [How to Convert String to Integer in Java](http://javarevisited.blogspot.com/2011/08/convert-string-to-integer-to-string.html). As these are common needs and having best way to do things in mind saves lot of time while coding.

### Convert Enum to String in Java Example

Enum classes by default provides valueOf (String value) method which takes a String parameter and converts it into enum. String name should match with text used to declare Enum in Java file. Here is a complete code example of String to Enum in Java

**Code Example String to Enum:**

**public** **class** **EnumTest** {

**private** **enum** LOAN {

HOME\_LOAN {

@Override

**public** String **toString**() {

**return** "Always look for cheaper Home loan";

}

},

AUTO\_LOAN {

@Override

**public** String **toString**() {

**return** "Cheaper Auto Loan is better";

}

},

PEROSNAL\_LOAN{

@Override

**public** String **toString**() {

**return** "Personal loan is not cheaper any more";

}

}

}

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

// Exmaple of Converting String to Enum in Java

LOAN homeLoan = LOAN.valueOf("HOME\_LOAN");

System.out.println(homeLoan);

LOAN autoLoan = LOAN.valueOf("AUTO\_LOAN");

System.out.println(autoLoan);

LOAN personalLoan = LOAN.valueOf("PEROSNAL\_LOAN");

System.out.println(personalLoan);

}

}

***Output:***

Always look **for** cheaper Home loan

Cheaper Auto Loan is better

Personal loan is not cheaper any more

### Convert Enum to String in Java Example

Now let's do opposite *convert an Enum into String in Java*, there are multiple ways to do it one way is to return exact same String used to declare **Enum from toString()** method of Enum, otherwise if you are using [toString() method](http://javarevisited.blogspot.com/2012/09/override-tostring-method-java-tips-example-code.html) for other purpose than you can use default [static](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) name() method to convert an Enum into String. Java by default adds name() method into every Enum and it returns exact same text which is used to declare enum in Java file.

**Code Example Enum to String**

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

// Java example to convert Enum to String in Java

String homeLoan = LOAN.HOME\_LOAN.name();

System.out.println(homeLoan);

String autoLoan = LOAN.AUTO\_LOAN.name();

System.out.println(autoLoan);

String personalLoan = LOAN.PERSONAL\_LOAN.name();

System.out.println(personalLoan);

}

*Output:*

HOME\_LOAN

AUTO\_LOAN

PERSONAL\_LOAN

That’s all on **How to parse String to Enum in Java and convert Enum to String object** . This tip will help you to quickly convert your data between two most versatile types Enum and String in Java. If you know any other way to change String to Enum in java than please let us know.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

http://javarevisited.blogspot.com/2011/07/jdk7-multi-cache-block-example-tutorial.html

How to use Java 1. 7 Multiple Catch Block with example - JDK 7 tutorial

As release of JDK 7 approaching General Availability (GA) on 2011/07/28, I thought to have a look on language enhancement as part of **project coin,** also called as **Small language enhancements or JSR 334**. Though there is not any major changes like [Enum](http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html) or [Generics](http://javarevisited.blogspot.com/2011/09/generics-java-example-tutorial.html) of Java 1.5,  but they are still very useful, in terms of simplifying your day to day programming task. Some of the interesting changes are allowing [String in Switch cases](http://javarevisited.blogspot.com/2011/08/string-switch-case-jdk7-example.html),  , inclusion of fork join framework in JDK itself , , type inference using diamond operator, automatic resource management using  [try with resource](http://javarevisited.blogspot.sg/2011/09/arm-automatic-resource-management-in.html) feature, and ability to catch multiple Exception in single catch block . In this Java 7 tutorial, we will learn how **multi catch block of JDK 1.7** makes Exception handling code simpler and elegant. Multiple catch block will allow you to catch multiple exceptions in one block but it’s only available in JDK7 and you need to compiler your code with source 1.7. This article also shows you how to use JDK 7 multiple catch block with example. I also recommend book Java 7 Recipes: A Problem-Solution Approach to learn more about all the changes made in JDK 1.7 and how to make effective use of them.

### JDK 1. 7 feature: Improved exception handling using multi-catch block

Java has always been criticized for having [checked exception](http://javarevisited.blogspot.sg/2011/12/checked-vs-unchecked-exception-in-java.html) and polluting code with cluttered exception handling code, multi-catch block in Java 1.7  certainly assuage those wounds. With **multi catch block,** you can catch multiple exceptions in one catch block, which will eventually result in more readable code. Prior to **JDK 7** if you want to catch two exceptions, you need to provide two catch blocks and if you have same code to run on these two blocks, then either you need to use [finally block](http://javarevisited.blogspot.sg/2012/11/difference-between-final-finally-and-finalize-java.html) or just duplicate the code on two catch blocks. finally block is  not an ideal solution, because it will execute even if Exception is not thrown so ultimately lot of duplicate code which sometime makes code unreadable and clumsy. Now with JDK7 multi catch block we can catch multiple exception in one catch block separated by pipe (|) and reduce the code duplication. Let’s see an example of multiple exceptions catching in Java 7.

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

    Scanner scnr = **new** Scanner(System.in);

    String number = scnr.next();

**try** {

**if** (number.length() > 5) {

**throw** **new** IllegalArgumentException();

        }

        Integer.parseInt(number);

    } **catch** (**NumberFormatException | IllegalArgumentException e**) {

        e.printStackTrace();

    }

}

In above code example or ***JDK7 multi-catch block*** we have used multiple catch block of JDK 1.7 and control will come on this block whenever code throws either NumberFormatException or IllegalArgumentException.

### Java 7  multiple catch block example tutorial

We have seen code making use of this new Java 7 feature of catching more than one Exception in one catch block. In our example, we are catching NumberFormatException and IllegalArgumentException together and her we will verify that by entering input which will result in both type of Exception one by one. If we are able to catch both Exception than it's proven.

### Testing of JDK 1.7 multi-cache block

### If we will enter any number with alphabets, than it will throw NumberFormatException as shown below :

Input: 23ff

**java.lang.NumberFormatException: For input string: "23ff"**

        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)

        at java.lang.Integer.parseInt(Integer.java:492)

        at java.lang.Integer.parseInt(Integer.java:527)

**at jdk7demo.JDK7Demo.main(JDK7Demo.java:25)**

Now let's enter number with more than 5 digit this will result in IllegalArgumentException as per our code.

Input :123333

**java.lang.IllegalArgumentException**

**at jdk7demo.JDK7Demo.main(JDK7Demo.java:23)**

I used Netbeans 7 to compile and run this project. **Setting up JDK 7 in Netbeans** is very easy just *download JDK7* and then click on **Tool-->Java Platform** and then click **"Add Platforms"** it will open a file browser just point out JDK7 installation directory and it will import JDK 1.7  binaries , source and docs and set it up for your use. One more thing you need to remember is that setting **source as 1.7** because this new language feature is only available in *JDK7*. In next series of this *JDK7 feature* article we will see [how to use String in Switch statement](http://javarevisited.blogspot.com/2011/08/string-switch-case-jdk7-example.html).

**Recommended Book for further Reading**

Java 7 Recipes: A Problem-Solution Approach By Josh Juneau, Carl Dea, Freddy Guime, John O'Conner

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2012/04/what-is-bounded-and-unbounded-wildcards.html>

What is bounded and unbounded wildcards in Generics Java?

**Bounded and un-bounded wildcards in Generics** are two types of wildcard available on Java. Any Type can be bounded either upper or lower of class hierarchy in Generics by using **bounded wildcards**. In short <? extends T> and <? super T> represent bounded wildcards while <?> represent an unbounded wildcard in generics . In our last article [How Generics works in Java](http://javarevisited.blogspot.com/2011/09/generics-java-example-tutorial.html) , we have seen some basic details about *bounded and unbounded wildcards in generics* and In this Java tutorial we will see bounded and unbounded generics wildcards in details. We will start from basics like *what is bounded wild card in generics* and *what is unbounded wildcard*  and than will some [popular java interview questions](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html) on generics like **Difference between ArrayList<? extends T>  and ArrayList<T super ?>.**

## What is bounded and unbounded wildcards in Generics

*bounded and unbounded wildcards in generics* are used to bound any Type. Type can be upper bounded by using <? extends T> where **all Types must be sub-class of T** or lower bounded using <? super T> where **all Types required to be super class of T**, here T represent lower bound. Single <?> is called an unbounded wildcard in generic and it can represent any type, similar to Object in Java. For example  **List<?>** can represent any List e.g. List<String> or List<Integer> its provides highest level of flexibility on passing method argument.

On the other hand **bounded wildcards provides limited flexibility within bound**. Any Type with bounded wildcards can only be instantiated within bound and any instantiation outside bound will result in compiler error.One of the important benefit of using bounded wildcard is that it not only **restrict number of Type** can be passed to any method as argument it also provides access to methods declared  by bound. for example TreeMap(Comparator<? super K> comparator) allows access to compare() method of [Comparator in Java](http://javarevisited.blogspot.com/2011/06/comparator-and-comparable-in-java.html).

## Example of Bounded and Unbounded wildcards in Java Generics:

Java Collection frameworks has several *examples of using bounded and unbounded wildcards in generics*. Utility method provided in Collections class accepts parametrized arguments. Collections.unmodifiableSet(Set<? extends T> s) and Collections.unmodifiableMap(Map<? extends K,? extends V> m) are written using  bounded wildcards which allows them to operate on either Collection of T or Collection of sub class or super class of T. just look at Java API for 1.5 and you will find lot of example of bounded and unbounded generic wildcards within JDK itself. If you are learning Java 1.5  you can also check [my post on Java Enum](http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html) and  [Variable arguments in Java](http://javarevisited.blogspot.com/2011/09/variable-argument-in-java5-varargs.html).

### When to use super and extends wildcards in Generics Java

Since there are two kinds of bounded wildcards in generics, **super and extends**, When should you use super wildcard and when should you extends wildcards. Joshua Bloch in Effective Java book has suggested **Producer extends, Consumer super** mnemonic regarding use of bounded wildcards. This book also has some good advice regarding how to use generics in Java and if you haven’t read it already, its worth reading book for Java programmer. Any way if type T is used as producer than use <? extends T>  and if type T represent consumer than use <? super T> bounded wildcards. Bounded wildcards in generics also increase flexibility of any API. To me its question of requirement, if a method also needs to accept any implementation of T then use extends wildcards.

### Difference between ArrayList<? extends T>  and ArrayList<? super T>

This is one of popular **generics interview question** , which is asked to check whether you are familiar to **bounded wildcards in generics**. both <? extends T> and <? super T> represent bounded wildcards, one will accept only T or sub class while other will accept T or super class. *bounded wildcards* gives more flexibility to methods which can operate on collection of T or its sub class. If you look at java.util.Collections class you will find several example of bounded wildcards in generics method. e.g. Collections.unmodifiableSet(Set<? extends T> s) will accept Set of type T or Set of sub class of T.

That's all on **what is bounded wildcards in generics**. both bounded and unbounded wild cards provides lot of flexibility on API design specially because Generics is not co-variant and List<String> can not be used in place of List<Object>. Bounded wildcards allows you to write methods which can operate on Collection of Type as well as Collection of Type subclasses.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.com/2011/09/variable-argument-in-java5-varargs.html>

Variable argument or Varargs methods from Java 5 with Example - Programming Tutorial

**Variable argument or varargs in Java** allows you to write more flexible methods which can accept as many argument as you need. variable arguments or varargswere added in Java 1.5 along with great language features like [Java Enum](http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html), [Generics](http://javarevisited.blogspot.com/2011/09/generics-java-example-tutorial.html), auto boxing and various others**.** Variable arguments a relatively small feature but useful for a developer who has been well aware about method and array**.** Some time we have a scenario that one method can take variable number of argument  and now with varargs from language makes it much easier. In this Java tutorial we will see How variable arguments makes it easy to write convenient method which can accept any number of arguments,  perfect candidates are sum() and average() kind of methods.

This article is in continuation of exploring features of Java programming language. I have already covered [fork-join framework from Java 7](http://javarevisited.blogspot.com/2011/09/fork-join-task-java7-tutorial.html) and [automatic resource management or ARM blocks](http://javarevisited.blogspot.com/2011/09/arm-automatic-resource-management-in.html), if you haven't read them already you may find them useful.

### Variable arguments before Java 1.5

Prior to Java 1.5 Java programmer mainly have two choices to :

1. Either overload the method.

2. Or can take an array or Collection and pass the no of argument wrapped in array or Collection like List, Set or Map.

But the problem with this is to if he is overloading the method and he don’t know about how many arguments he has to handle how many method will be created in the code i.e the code will become clumsy or if he has not created sufficient method then again the codes need to be modified and complied so it’s become repetitive task which is not a good [programming practice](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html) and requires more maintenance .Now we can go for array also but ks why not we give this task to Java for creating an array and store the element in to that array to internally handle  and let make programmer free of this, I guess with this thought  **varargs** comes into existence.

varargs or variable arguments makes it possible for us to call one method with variable number of argument; means define only one method and call that method with zero or more than zero argument.

**Syntax:**

**type … variable Name.**

Ellipses stands for variable argument java treats variable argument as an array of same data type. 3 dots is used to denote variable argument in a method and if there are more than one parameter, varargs arguments must be last, as better listed below

**Some points which should be taken care when use varargs:**

1. Ellipse can be used once in method parameter list.
2. Ellipse with type must be used in parameter list at the end of the method

### Real world Example of varargs in Java

First we look one real world scenario suppose we go one college and take admission on that college now its not really decided that admission will be done for how many student may be 50 student will come or 100 or more than that at a time. So college is one class and Admission is one procedure or method that takes no of student as an argument .So in that method we can use varargs or variable arguments.

***/\*\****

***\* Simple real world example of variable argument methods***

***\*/***

**public** **class** college {

**public** **void** admission\_method (**int**... no\_of\_student) {

*//rest of code for processing*

}

}

**Simple java variable argument example:**

Let consider one simple example of finding the multiplication of n number. First we will try to solve this problem using method overloading

***/\*\****

***\* Java Program which tries to implement variable argument method using***

***\* method overloading. This started get clumsy once number of parameter exceeds***

***\* five.***

***\*/***

**class**  VarargsExample{

**public** **int** multiply(**int** a,**int** b){ **return** a\*b;}

**public** **int** multiply(**int** a,**int** b,**int** c){ **return** (a\*b)\*c;}

**public** **int** multiply(**int** a,**int** b,**int** c,**int** d{ **return** (a\*b)\*(c\*d);}

}

If we use method overloading same method will be repeated again and again and its not worth after four or five parameters. now will use array also to solve this problem of variable arguments:

Let see how:

***/\*\****

***\* Java Program which tries to implement variable argument method using***

***\* method overloading. This started get clumsy once number of parameter exceeds***

***\* five.***

***\*/***

**class**  VarargsExample{

*/\**

*\* @return multiplication of all numbers in array*

*\*/*

**public** **int** multiply(**int**[] numbers){

**int** result = 1;

**for**(**int** number: numbers){

      result= result\*number;

    }

**return** result

  }

}

Here we need to create an integer array and  pass that array to the method and then iterate the array and get result .

We can simplify this with **variable argument provided by java 5** where creation of array will be done internally and our task become easier.

***/\*\****

***\* Java Program which uses varargs feature to accept variable number of***

***\* arguments. variable arguments are implemented using anonymous array so if***

***\* another method with exact same signature except array in place of varargs will result***

***\* in compiler error.***

***\*/***

**class**  VarargsExample{

*/\**

*\* @ return multiplication of all numbers in array*

*\* if varargs method accept more than one parameter than varargs arguments*

*\* must be last parameter.*

*\*/*

**public** **int** multiply(**int**... numbers){

**int** result = 1;

**for**(**int** number: numbers){

      result= result\*number;

    }

**return** result

  }

}

## Important points related to variable argument or varargs methods:

1) Every call to **varargs method** require an [anonymous array](http://javarevisited.blogspot.com/2012/01/anonymous-array-example-java-create.html) to be created and initialized which could affect performance in time critical application. There is an alternative of varargs method to achieve better performance. suppose you have a variable argument method sum(int... num) and its called with 2 parameters on 90% of time. In order to avoid array creation and initialization you can use [method overloading in Java](http://javarevisited.blogspot.com/2011/12/method-overloading-vs-method-overriding.html) to provide two versions of sum() which accept int instead of varargs. here is an example of better performance alternative of varargs for 90% of time

public int sum(int a);

public int sum(int a, int b);

public int sum(int... num);

Now 90% of time **method without varargs will be invoked** and 10% of time method with variable argument will be invoked.

2) An **example of variable argument method from JDK** is Arrays.asList(T... args) which was used to [convert array to ArrayList](http://javarevisited.blogspot.com/2011/06/converting-array-to-arraylist-in-java.html) before JDK 1.5 but retrofitted to support variable argument in JDK 1.5. Now you can also invoke this method by just passing as many Strings or object as you want and creating a List representation on the fly. Its one of the quickest way to [convert Strings into List](http://javarevisited.blogspot.com/2011/06/converting-array-to-arraylist-in-java.html) e.g.

List listOfString = Arrays.asList("Red", "White", "Blue");

3) Another example of varargs methods are in java.lang.reflect package. Reflection uses lot of variable argument method to [call overloaded method dynamically](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html). Method class used variable argument to get correct version of overloaded method. Method.getMethod(String name, Class... parameterTypes) uses last argument as parameter type which is a variable argument and can accept any number of parameters. This is used to invoke method by name using reflection.

4) If you are working on a legacy project which is not running on Java 1.5 or higher, you can still implement variable argument methods by using **Anonymous array** or **Collection classes** like [ArrayList or HashSet](http://javarevisited.blogspot.com/2012/01/convert-arraylist-to-set-java-example.html). Both array or Collection classes can wrap number of argument into one. Using Collection framework also has an added advantage in terms of rich API e.g. meaningful toString() method, iteration support etc.

That’s all on variable arguments or varargs in Java, Please let me know how you guys use variable arguments and what your opinion about it is.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2012/06/10-interview-questions-on-java-generics.html>

10 Interview Questions on Java Generics for Programmer and Developers

**Generic interview questions** in Java interviews are getting more and more common with Java 5 around there for considerable time and many application either moving to Java 5 and almost all new Java development happening on Tiger(code name of Java 5).  Importance of Generics and Java 5 features like Enum,  Autoboxing, varargs and Collection utilities like CountDownLatch, CyclicBarrier and BlockingQueue are getting more and more popular on Java interviews. Generic interview question can get real tricky if you are not familiar with [bounded and unbounded generic wildcards](http://javarevisited.blogspot.com/2012/04/what-is-bounded-and-unbounded-wildcards.html), **How generics works internally**, type erasure and familiarity with writing parametrized generics classes and methods in Java. Best way to prepare for Generics interview is to try simple program best on various features of generics. Anyway In this Java interview article we will see some popular Java Generics interview questions and there answer. By the way there are lot of material available in Javarevisited to better preparing for Java and J2EE interviews, you can prepare multi-threading and Collections using 15 thread interview question and [Top 10 Java collection interview question](http://javarevisited.blogspot.sg/2011/11/collection-interview-questions-answers.html) along with several other questions answers articles on Spring, Struts, JSP and Servlet. If you are GUI developers and working in Java Swing technology than you can also check interview questions on Java Swing  mostly asked in Investment banks

## Java Generics Interview Questions

**1. What is Generics in Java ? What are advantages of using Generics?**

This is one of the first interview questions asked on generics in any Java interview, mostly at beginners and intermediate level. Those who are coming from prior to Java 5 background knows that how inconvenient it was to [store object in Collection](http://javarevisited.blogspot.com/2012/07/create-read-only-list-map-set-example-java.html) and then cast it back to correct Type before using it. Generics prevents from those. it provides [compile time](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html) type-safety and ensures that you only insert correct Type in collection and avoids ClassCastException in runtime.

**2. How Generics works in Java ? What is type erasure ?**

This is one of better interview question in Generics. Generics is implemented using **Type erasure**, compiler erases all type related information during compile time and no type related information is available during runtime. for example List<String> is represented by only [List](http://www.blogger.com/goog_1304192070)at runtime. This was done to ensure binary compatibility with the libraries which were developed prior to Java 5. you don't have access to Type argument at runtime and **Generic type** is translated to **Raw type** by compiler during [runtime](http://javarevisited.blogspot.sg/2012/03/what-is-static-and-dynamic-binding-in.html). you can get lot of follow up question based on this Generic interview question based upon your response e.g. *Why Generics is implemented using Type erasure* or presenting some invalid generic code which results in compiler error. read my post [How Generics works in Java](http://javarevisited.blogspot.com/2011/09/generics-java-example-tutorial.html) for more details

**3. What is Bounded and Unbounded wildcards in Generics ?**

This is another very [popular Java interview questions](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html) on Generics. Bounded Wildcards are those which impose bound on Type. there are two kinds of Bounded wildcards <? extends T> which impose an upper bound by ensuring that type must be sub class of T and <? super T> where its imposing lower bound by ensuring Type must be super class of T. This Generic Type must be instantiated with Type within bound otherwise it will result in compilation error. On the other hand <?> represent and unbounded type because <?> can be replace with any Type. See more on my post [differences between Bounded and Unbounded wildcards in Generics](http://javarevisited.blogspot.com/2012/04/what-is-bounded-and-unbounded-wildcards.html).

**4. What is difference between List<? extends T>  and  List <? super T> ?**

This is related to previous generics interview questions, some time instead of asking what is bounded and unbounded wildcards interviewer present this question to gauge your understanding of generics. Both of List declaration is e[xample of bounded wildcards,](http://javarevisited.blogspot.sg/2012/04/what-is-bounded-and-unbounded-wildcards.html) List<? extends T> will accept any List with Type extending T while List<? super T> will accept any List with type super class of T. for Example List<? extends Number> can accept List<Integer> or List<Float>. see more on above link.

**5. How to write a generic method which accepts generic argument and return Generic Type?**

writing generic method is not difficult, instead of using raw type you need to use Generic Type like T, E or K,V which are well known placeholders for Type, Element and Key, Value. Look on [Java Collection framework](http://javarevisited.blogspot.sg/2011/11/collection-interview-questions-answers.html) for examples of generics methods. In simplest form a generic method would look like:

**public** V put(K key, V value) {

**return** cache.put(key, value);

}

**6. How to write parametrized class in Java using Generics ?**

This is an extension of previous Java generics interview question. Instead of asking to write Generic method Interviewer may ask to *write a type safe class using generics*. again key is instead of using raw types you need to used generic types and always use standard place holder used in JDK.

**7. Write a program to implement LRU cache using Generics ?**

This is an exercise for anyone who like [Coding in Java](http://javarevisited.blogspot.sg/2011/09/code-review-checklist-best-practice.html). One hint is that LinkedHashMap can be used implement fixed size LRU cache  where one needs to remove eldest entry when Cache is full. LinkedHashMap provides a method called removeEldestEntry() which is called by put() and putAll() and can be used to instruct to remove eldest entry. you are free to come up with your own implementation as long as you have a written a working version along with [JUnit test](http://javarevisited.blogspot.sg/2012/06/junit4-annotations-test-examples-and.html).

**8. Can you pass List<String> to a method which accepts List<Object>**

This generic interview question in Java may look confusing to any one who is not very familiar with Generics as in fist glance it looks like String is object so List<String> can be used where List<Object> is required but this is not true. It will result in compilation error. It does make sense if you go one step further because List<Object> can store any any thing including [String, Integer](http://javarevisited.blogspot.com/2011/08/convert-string-to-integer-to-string.html) etc but List<String> can only store Strings.

**List**<**Object**> objectList;

**List**<**String**> stringList;

objectList = stringList;  *//compilation error incompatible types*

**9. Can we use Generics with Array?**

This was probably most simple generics interview question in Java, if you know the fact that Array doesn't support Generics and that's why Joshua Bloch suggested in Effective Java to prefer List over Array because *List can provide compile time type-safety* over Array.

**10. How can you suppress unchecked warning in Java ?**

javac compiler for Java 5 generates unchecked warnings if you use combine raw types and generics types e.g.

**List**<**String**> rawList = **new** **ArrayList**()

Note: Hello.java uses unchecked or unsafe operations.;

which can be suppressed by using @SuppressWarnings("unchecked") annotation.

**Java Generics Interview questions Update:**

I got few more interview questions on Generics in Java to share with you guys, These questions focus on What is difference between Generics type and Raw type and Can we use Object in place of bounded wildcards etc:

**Difference between List<Object> and raw type List in Java?**

Main difference between raw type and parametrized type List<Object> is that, [compiler](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html) will not check type-safety of raw type at compile time but it will do that for parametrized type and by using Object as Type it inform compiler that it can hold any Type of Object e.g. String or Integer. This Java Generics interview question is based on correct understanding of raw type in Generics. Any way second difference between them is that you can pass any parametrized type to raw type List but you can not pass List<String> to any method which accept List<Object> it will result in compilation error. Read [How Generics works in Java](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html) for more details.

**Difference between List<?> and List<Object> in Java?**

This generics interview question may look related to previous interview questions but completely different. *List<?> is List of unknown type* while *List<Object> is essentially List of any Type*. You can assign List<String>, List<Integer> to List<?> but you can not assign List<String> to List<Object>.

**List**<?> listOfAnyType;

**List**<**Object**> listOfObject = **new** **ArrayList**<**Object**>();

**List**<**String**> listOfString = **new** **ArrayList**<**String**>();

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

listOfAnyType = listOfString; *//legal*

listOfAnyType = listOfInteger; *//legal*

listOfObjectType = (**List**<**Object**>) listOfString; *//compiler error - in-convertible types*

to know more about wildcards see [Generics Wildcards Examples in Java](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html)

**Difference between List<String> and raw type List.**

This Generics interview question is similar to difference between raw type and parametrized type.Parametrized type are type-safe and type-safety will be guaranteed by compiler but [List raw type is not type safe](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html). You can not store any other Object on List of String but you can not store any Object in raw List. There is no casting required in case of Parametrized type with Generics but explicit casting will be needed for raw type.

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

listOfRawTypes.add("abc");

listOfRawTypes.add(123); *//compiler will allow this - exception at runtime*

**String** item = (**String**) listOfRawTypes.get(0); *//explicit cast is required*

item = (**String**) listOfRawTypes.get(1); *//ClassCastException because Integer can not be cast in String*

**List**<**String**> listOfString = **new** **ArrayList**();

listOfString.add("abcd");

listOfString.add(1234); *//compiler error, better than runtime Exception*

item = listOfString.get(0); *//no explicit casting is required - compiler auto cast*

These were some of the *frequently asked generics interview questions and answers in Java*. None of these generic interview questions are tough or hard, Indeed they are based on fundamental knowledge of generics. Any Java programmer who has decent knowledge of Generics must be familiar with these generics questions in Java. If you have any other good generic question which has been asked in any interview or you are looking answer for any Generics interview question in Java then please post in comment section.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.sg/2011/09/arm-automatic-resource-management-in.html>

ARM- Automatic resource management in Java7 an example tutorial

ARM **automatic resource management** is  another attractive features of Java 7 and project coin. Asname itself implies that now JVM is going to be handling all the external resource and make programmer free to bother about resource management.  If my java programmers use any external resources like file, printer or any devices to close after my program execution complete. Normally we close the resources which we have open in beginning of our program or we decide that if program finish normally how to manage the resource or if our program finish abnormally how to close the resource.

## ARM- Automatic resource management in Java

Earlier this things or **resource management** are achieved **by try {} catch {} finally block** we need to declare resource variable outside of try block and in catch or finally we use this variables for achieving normal execution of program. This JDK7 tutorial is in continuation of my earlier tutorial [How to code with multi-cache exception in JDK7](http://javarevisited.blogspot.com/2011/07/jdk7-multi-cache-block-example-tutorial.html) and [How to use String in Switch case on JDK7](http://javarevisited.blogspot.com/2011/08/string-switch-case-jdk7-example.html)

### Example of resource management in java before JDK7

Here is an example of how we used to do handle *resource management* before [automatic resource management (ARM) feature](http://openjdk.java.net/projects/jdk7/features/) was made available.

FileInputStream stockQuoteReader= **null**;

      FileOutputStream stockQuoteWriter = **null**;

**try** {

        stockQuoteReader = **new** FileInputStream("StockQuotes.txt");

        stockQuoteWriter = **new** FileOutputStream("StockQuotes.txt");

**int** var;

**while** (var = stockQuoteReader.read()) != -1)

          stockQuoteWriter.write(var);

      } **finally** {

**if** (stockQuoteReader!= **null**)

          stockQuoteReader.close();

**if** (stockQuoteWriter!= **null**)

          stockQuoteWriter.close();

      }

But with java 1.7 we manage this thing very easily by **try with resource block** where inside try we mange this external resources.

### Signature of Automatic Resource Management (ARM)

**Signature** is **try(resource1;resource2){}**after final resource **;semicolon is not allowed** and the resource should be like **var=expression type** and bydefault all the resources are final  type.

### What has been added in API for Automatic Resource Management

**java.lang.AutoCloseable**, interface has been added in API which contains single method **close() throws Exception** this interface is a parent of **java.io.closeable** interface so all the input and output devices inherit this property.

### Example of Automatic Resource Management (ARM) in JDK7

Here is **example of automatic resource management with JDK 1.7** source base. Please make sure you run this with java source 1.7 otherwise you will get compilation error.

**try** (

FileInputStream stockQuoteReader = **new** FileInputStream("StockQuotes.txt");

FileOutputStream stockQuoteWriter = **new** FileOutputStream("StockQuotes.txt")

) {

**int** var;

**while**((var= stockQuoteReader.read()) != -1 )

            stockQuoteWriter.write();

  }

In this code inside try we have declare two file stream one is input file we are reading from one file and writing to another file. After the whole process both streams will be closed automatically either the code has been executed normally or not that means **stockQuoteReader.close()** and stockQuoteWriter.close() called automatically which is the best part of ARM.

If we compare this with earlier example   then if any exception occur during input file closing i.e. **stockQuoteReader.close()** , stockQuoteWriter.close() will never get executed so our code terminated abnormally.

### Some important points which needs to be keep in mind when use ARM

          Whatever resource we are using should be subtypes ofAutoCloseable other wise will get compile time error.

          The resources which we are using are closed in reverse order means stockQuoteWriter.close() will be called first then stockQuoteReader.close().

That’s all on **new automatic resource management (ARM) feature on JDK7,** some how it address the cluttering of code due to checked exception handling and code duplication on several exception cache block.