TreeMap

Eg: TreeMap\_Eg1

Consists of key and value ,comes with sorted.

duplicates not allowed

in treemap can used for complex object ( like student data , employee data )

IdentityHashMap:

Features of IdentityHashMap

* It follows reference equality, instead of using the equals() method . it uses the == operator.
* It is not synchronized and must be synchronized externally.
* Iterators are fail-fast, throw ConcurrentModificationException in an attempt to modify while iterating.
* works similarly as hashmap

Enumerations :

It is introduced in java 1.5 . 98 % percent you will not use , only 2 percent we might use it . but internally it is used in the annotation

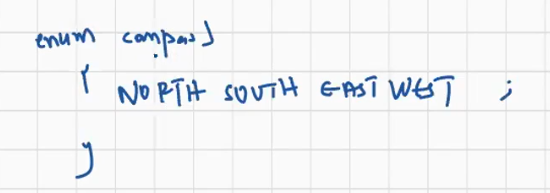
Constant : if a variable is marked as final then it is called as constant . it is industry recommended to give the constant variable name in upper case

enum – group of named / predefined constants

Constant value during declaration it is decided it cannot be changed later.

also its refers to enumerated data / our own data type

whenever there is a group of similar predefined constants we go with concept enum.



North etc are compass type data / our own data type .

To create enum we use keyword enum followed by name

syntax:

enum Something {

;;;;

;;;;

}

enum Day{

SUN; // by default constants they are static final

MON; // they should be capital industry recomended

TUE;

}

// the above constants are of day type data

Here constants are value / data , ref var also .

Eg: SUN // ref var , data / value

Outside class we can define enum

Inside also we can define enum

Class can also be declared inside the enum

Eg: Enum\_Inside\_Outside\_Class

For every interface and class a separate .class is generated

Separate .class for enum is also created

Inside enum we can have

fields (instance variables) , methods, main method , constructor. But only after declaring the constants ( or) , followed by semi colon .

Eg: Variables\_Methods\_Constructor\_Inside\_Enum

Constant itself is a variable and a value

Eg: Constant\_Itself\_Variable\_And\_Value

Eg: Default\_Syntax\_Of\_Constant\_And\_Constructor\_Calling

Inside enum don’t make constructor as public , only default and private are allowed

We can use constant reference to call the inbuilt methods of the enum class

Eg: Built\_In\_Methods

Setter and getter in enum

Eg: Setter\_Getter\_In\_Enum

// go through the code

Annotation

In programming, comments are a way to add notes and explanations to your code.

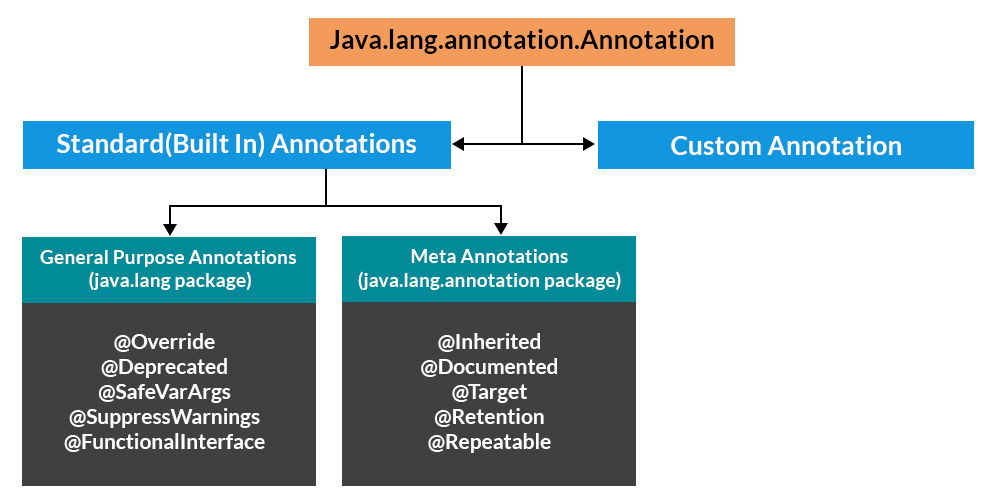
Comments are only for developer will not be a part in compilation.

Annotations in [Java](https://www.simplilearn.com/tutorials/java-tutorial/what-is-java) provide additional information to the compiler and [JVM](https://www.simplilearn.com/jvm-and-the-implications-of-sandbox-model-rar41-article).

If you want your message to be given to compiler , jvm and we use annotations . just like comments give some message to developers

An annotation is a tag representing metadata about [classes](https://www.simplilearn.com/tutorials/java-tutorial/java-classes-and-objects), [interfaces](https://www.simplilearn.com/tutorials/java-tutorial/java-interface), variables, [methods](https://www.simplilearn.com/tutorials/java-tutorial/methods-in-java) (or) fields. Annotations do not impact the execution of the code that they annotate. Some of the characteristics of annotations are:

* Begin with ‘@’
* Do not alter the execution of the program
* Provide supplemental information and help to link metadata with elements of a program such as classes, variables, constructs, methods, etc.
* Annotations are not pure comments as they can change the way a program is treated by the compiler.



Eg: Annotation\_Eg1

// go through the code

Eg: Annotation\_Eg2

// go through the code

Annotation can be applied to

1. Class
2. interface
3. method
4. fields ( instan var )
5. local vars
6. constructor
7. for parameters
8. enums

Annotation is introduced from java 5

To create a user defined annotation use interface with @

Category 1: Marker Annotations

The marker annotation is used to mark a declaration. It does not include any members or data. Only the presence of a marker annotation as an annotation is sufficient. An example of Marker Annotation is @Override

Example

@TestAnnotation()

### Category 2: Single value Annotations

A single value annotation consists of a single member only. This annotation allows a shorthand form to specify the value of the member. There is no need to specify the name of the member, only the value of the member has to be specified.

#### Example

#### Creating a custom exception to show single value annotation

#### to create a custom exception we use interface with @

#### Inside a interface there are values . There are 2 ways to assign them

@interface AnnotationName{

int run() default 0; // by using default and passing the value

}

2nd way : To apply single value annotation, use

@AnnotationName(value=6)

### Category 3: Full Annotations

Full Annotations include multiple data members, values, names, and pairs.

#### Example

@TestAnnotation(owner= ”Ravi”, value= ”Class ”)

Eg: Multi\_Full\_Value\_Annotation

Whenever we are creating the custom annotation we need to specify two things

1. Every annotation is applied at specific level like class , constructor , method , variable , parameter etc.

How to specify ?

One meta annotation is there , means annotation you are using for another annotation.

@Target

1. Retention policy : specifies whatever annotation we are creating , that extra information should reach the either till compiler ( or) jvm

Eg: Target\_Retention

98% we are not using custom annotations , mostly uses predefined annotations .

To get values of annotation can use reflection api

Get annotation not get annotation very often

Enums can be passed to switch case

Generics :

Def: The main objective of Generics is to provide Type-Safety and to resolve Type-Casting problems.

Case 1: Type-Safety

Arrays are always type safe that is we can give the guarantee for the type of elements present inside array.

For example if our programming requirement is to hold String type of objects it is recommended to use String array. In the case of string array we can add only string type of objects by mistake if we are trying to add any other type we will get compile time error.

eg: Type\_Safety

That is we can always provide guarantee for the type of elements present inside array and hence arrays are safe to use with respect to type that is arrays are type safe.

But collections are not type safe that is we can't provide any guarantee for the type of elements present inside collection.

For example if our programming requirement is to hold only string type of objects it is never recommended to go for ArrayList. By mistake if we are trying to add any other type we won't get any compile time error but the program may fail at runtime.

Eg: Type\_Casting\_Eg1

Hence we can't provide guarantee for the type of elements present inside collections that is collections are not safe to use with respect to type.

Case 2: Type-Casting

In the case of array at the time of retrieval it is not required to perform any type casting.

Eg:: Type\_Casting\_Array\_During\_Retrival

But in the case of collection at the time of retrieval compulsory we should perform type casting otherwise we will get compile time error.

Eg :: TypeCasting\_Collection\_During\_Retrival

That is in collections type casting is bigger headache.

To overcome the above problems of collections (type-safety, type casting) sun people introduced generics concept in 1.5v

hence the main objectives of generics are:

1. To provide type safety to the collections.

2. To resolve type casting problems.

To hold only string type of objects we can create a generic version of ArrayList as follows.

Eg: Type\_Safety\_With\_Genrics

For this ArrayList we can add only string type of objects by mistake if we are trying to add any other type we will get compile time error that is through generics we are getting type safety.

At the time of retrieval it is not required to perform any type casting we can assign elements directly to string type variables.

Eg: Type\_Casting\_With\_Genrics

That is through generic syntax we can resolve type casting problems.

Conclusions

1. Polymorphism concept is applicable only for the base type but not for parameter type

[usage of parent reference to hold child object is called polymorphism].

List<String> al =new ArrayList<String>();

eg: ArrayList<String> al =new ArrayList<String>();

here ArrayList the base paremeter and String is the type parameter

Collection<String> al =new ArrayList<String>();

Collection<Object> al =new ArrayList<String>(); //CE:incompatible types

type parameters can’t be written based on the parent child relations as shown above.

1. Collection concept applicable only for objects, Hence for the parameter type we can use any class or interface name but not primitive value(type). Otherwise we will get compile time error.

eg: ArrayList<int> al =new ArrayList<int>();

//CE: unexcpected type found :primitive required: reference

Generic classes:

Until 1.4v a non-generic version of ArrayList class is declared as follows.

Example:

class ArrayList{

add(Object o);

Object get(int index);

}

add() method can take object as the argument and hence we can add any type of object to the ArrayList.

Due to this we are not getting type safety.

The return type of get() method is object hence at the time of retrieval compulsory we should perform type casting.

But in 1.5v a generic version of ArrayList class is declared as follows.

|=> Type parameter

class ArrayList<T>{

add(T t);

T get(int index)

}

Based on our requirement T will be replaced with our provided type.

For Example to hold only string type of objects we can create ArrayList object as follows.

Example:

ArrayList<String> I=new ArrayList<String>();

For this requirement compiler considered ArrayList class is

Example:

class ArrayList<String>{

add(String s);

String get(int index);

}

add() method can take only string type as argument hence we can add only string type of objects to the List.

By mistake if we are trying to add any other type we will get compile time error.

eg#1

ArrayList<String> al =new ArrayList<String>();

al.add("Pavan kalyan");

al.add(10);//CE: can't find symbol

symbol: method add(int)

location : class java.util.ArrayList<java.lang.String>

al.add(10)

eg#2.

ArrayList<String> al =new ArrayList<String>();

al.add("NavinReddy");

String name = al.get(0); //type casting is not requried

Hence through generics we are getting type safety.

At the time of retrieval it is not required to perform any type casting we can assign its values directly to string variables.

Normal class : A class without any type parameter

Generic class : A class with type parameter

In generics we are associating a type parameter to the class , such type of parameterized classes are nothing but generic classes

Based on our requirement we can create our own generic classes also.

Example:

class Account<T>

Account<Gold>g1 = new Account<Gold>();

Account<Silver> g2 = new Account<Silver>();

Eg: User\_Defined\_Generics

Note: to get the underlying object type of any reference we use a method called ref.getclass().getname()

eg1

interface Calculator{}

class Casio implements Calculator{}

class Quartz implements Calculator{}

class Kadio implements Calculator{}

Calculator c1 =new Casio();

System.out.println(c1.getClass().getName()); //Casio

// reference type is calculator but the runtime object is casio. to get the runtime object we use this.

Calculator c2 =new Quartz();

System.out.println(c2.getClass().getName()); //Quartz

Calculator c3 =new Kadio();

System.out.println(c3.getClass().getName()); //Kadio

Bounded types :

We can bound the type parameter for a particular range ( we can give only a particular type parameter ) by using extends keyword such types are called bounded types.

Example 1:

class Test<T>

{}

Test <Integer> t1=new Test< Integer>();

Test <String> t2=new Test < String>();

Here as the type parameter we can pass any type and there are no restrictions hence it is unbounded type.

Example 2:

class Test<T extends X>

If x is a class then as the type parameter we can pass either x or its child classes.

If x is an interface then as the type parameter we can pass either x or its implementation classes.

Eg1 :

class Test <T extends Number>{}

class Demo{

public static void main(String[] args){

Test<Integer> t1 = new Test<Integer>();

Test<String> t2 = new Test<String>(); //CE

}

}

We can give only Number types i.e Byte , Short , Integer , Long , Float , Double etc.

eg#2.

class Test <T extends Runnable>{}

class Demo{

public static void main(String[] args){

Test<Thread>t1 = new Test<Thread>();

Test<String> t2 = new Test<String>(); //CE

}

}

Keypoints about bounded types :

=> We can't define bounded types by using implements and super keyword

=> even if use interface in the type parameter we should use extends keyword not implements.

eg:

class Test<T implements Runnable>{} //invalid

class Test<T super String>{} //invalid