**HashMap**

**Recap:**

1]The methods get, put, remove work in constant time i.e. O(1).

2]Insertion order is not preserved

3]Permits one null key, null values

4]Methods are not synchronized unlike Hash Table

**Demo**

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** HashMapDemo1 {

**public** **static** **void** HashMapdemo() {

Map <String,Integer> map1= **new** HashMap<>();

map1.put("Raj", 25);

map1.put("John", 24);

map1.put("Anitha",**null**); //null values are allowed

System.***out***.println(map1);

map1.put("Anitha",20); // value corresponding to Anitha will be over written

System.***out***.println(map1);

System.***out***.println(map1.containsKey("Raj")); //checks whther the given key exists

System.***out***.println("age: "+map1.get("John")); //returns value for the given key

Set<String> names= map1.keySet(); //get list of keys - way I of iteration

**for**(String name:names) {

System.***out***.println("name "+name +" age: "+ map1.get(name));

}

Set <Map.Entry<String, Integer>> set1= map1.entrySet(); //2nd way of iteration

**for**(Map.Entry<String, Integer> mapping: set1) {

System.***out***.println(mapping.getKey()+" "+mapping.getValue() );

}

names.remove("John"); //changes in names will be reflected in map

System.***out***.println(map1);

Map<String,Map<String,Object>> userProfile=**new** HashMap<>(); //nested map

Map<String,Object> profile=**new** HashMap<>();

profile.put("age", 34);

profile.put("branch", "CS");

profile.put("city", "Mysuru");

userProfile.put("Ramya", profile);

profile=**new** HashMap<>();

profile.put("age", 38);

profile.put("branch", "CS");

profile.put("city", "Mysuru");

userProfile.put("Raa", profile);

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

Map<String,Object> user1=userProfile.get("Raa");

**int** age=(Integer)(user1.get("age")); //type casting since user1.getAge() has the return type object

System.***out***.println("äge of Raa " + age);

}

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

*HashMapdemo*();

}

}

**Output**

{John=24, Raj=25, Anitha=null}

{John=24, Raj=25, Anitha=20}

true

age: 24

name John age: 24

name Raj age: 25

name Anitha age: 20

John 24

Raj 25

Anitha 20

{Raj=25, Anitha=20}

{Raa={city=Mysuru, branch=CS, age=38}, Ramya={city=Mysuru, branch=CS, age=34}}

äge of Raa 38

**Beware of creating mutable keys**

**Initial code:**

**public** **static** **void** immutableKeys() {

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

list.add(1);

Map <List<Integer>,Integer> map=**new** HashMap<>();

map.put(list, 1);

list.add(2);

System.***out***.println(map.get(list));

}

**Output**

Null

Explanation:

1]When map.put(list,1) is called the HashMap invokes HashCode method on the key.

The HashCode method would return a hashCode

2]In list.add(2), the hashCode method is invoked again. It returns a completely different hashCode.

3]ArrayList overrides AbstractList. In AbstarctList, the “HashCode” method is overridden.

**Code: public** **int** hashCode() {

**int** hashCode = 1;

**for** (E e : **this**)

hashCode = 31\*hashCode + (e==**null** ? 0 : e.hashCode());

**return** hashCode;

}

Acc to code, each element of list is iterated to compute the hashCode. Hence, the hashCode computed will be different in both the cases.

Ex 2:

**public** **class** Student {

**public** **int** rollno;

**public** String name;

**public** **int** getRollno() {

**return** rollno;

}

**public** **void** setRollno(**int** rollno) {

**this**.rollno = rollno;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Student(**int** rollno, String name){

rollno=**this**.rollno;

name=**this**.name;

}

}

Map<Student,Integer> map1=**new** HashMap<>();

Student s=**new** Student(1,"Ram");

map1.put(s, 1);

s.setName("John");

System.***out***.println(map1.get(s));

}

**Output**

**1**

Explanation:

Student class uses hashCode and equals method of Object class. These methods in Object class compute hashCode based on memory address of objects.

**LinkedHashMap**

* LinkedHashMap preserves insertion order without affecting the access speed [It has DoublyLinkedList running through it’s entries]
* Map-> HashMap -> LinkedHashMap
* It is LinkedList and HashTable implementation of Map interface.
* It supports lookup, insertion and deletion in O(1).
* Just like HashMap – It allows one null key, null values
* Like HashMap – the methods are not synchronised.
* linkedHashMap is faster than HashMap when it comes to iterating the elements. Because, iteration is dependant on size of the map. Whereas, in HashMap – the iteration is based on capacity of the map not the actual size.

LinkedHashMap as LRU cache [Least Recently used Cache]

* Cache means the important data is stored in RAM for faster access.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

Orange- least recently used item in cache

Green-most recently used item in cache

When cache is full, least recently used item is deleted from cache. When new item is accessed, it will be added as most recently used item of cache. If already existing element in cache is accessed, it will be moved as most recently used element of cache.

* The LRU cache size in case of LinkedHashMap will be unlimited.

So, to remove least recently used element, we will have to create a subclass and override the method removeEldestEntry(Map.Entry<K,V> eldest) and specify the proper size of cache. If cache is full, the ldest entry will be removed.

Code

**public** **static** **void** cacheDemo() {

Map<String, String> lruCache = **new** LinkedHashMap<> (16, 0.75f, **true**); //should be true if we want caching

lruCache.put("a", "A");

lruCache.put("b", "B");

lruCache.put("c","C");

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

lruCache.get("a");

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

lruCache.get("b");

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

lruCache.put("d", "D");

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

}

Output

{a=A, b=B, c=C}

{b=B, c=C, a=A}

{c=C, a=A, b=B}

{c=C, a=A, b=B, d=D}

Ex 2:

LinkedHashMap is subclassed and ‘true” is returned from removeEldestEntry if size> some fixed value

**public** **static** **void** cacheDemo() {

Map<String, String> lruCache = **new** LruCache<>(16, 0.75f, **true**);

lruCache.put("a", "A");

lruCache.put("b", "B");

lruCache.put("c", "C");

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

lruCache.get("a"); // multiple gets to "a" will not make a difference

lruCache.get("a");

lruCache.get("a");

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

lruCache.get("b");

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

lruCache.put("d", "D");

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

lruCache.put("e", "E");

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

}

import java.util.LinkedHashMap;

import java.util.Map;

public class LruCache<K,V> extends LinkedHashMap<K,V> {

private static final int MAX\_ENTRIES = 3;

public LruCache(int initialCapacity,

float loadFactor,

boolean accessOrder) {

super(initialCapacity, loadFactor, accessOrder);

}

// Invoked by put and putAll after inserting a new entry into the map

public boolean removeEldestEntry(Map.Entry eldest) {

return size() > MAX\_ENTRIES;

// return false; // same as normal linked hash map

}

}

Output

{a=A, b=B, c=C}

{b=B, c=C, a=A}

{c=C, a=A, b=B}

{a=A, b=B, d=D}

{b=B, d=D, e=E}

**SortedMap, NavigableMap, TreeMap**

* Map-> SortedMap-> NavigableMap-> TreeMap
* Sorted by keys – 2 types of sorting

A]natural ordering [implement Comparable interface]

B]Comparator

Public Interface SortedMap<K,V> extends Map<K,V>{

//range view

SortedMap<K,V> subMap(K fromKey, K to Key)

SortedMap<K,V> headMap(K to Key)

SortedMap<K,V> tailMap(K fromKey, K to Key)

//end points

K firstKey()

K Lastkey()

//comparator access

Comparator<?superE> comparator()

//collection-view

Set<K> keySet()

Collection<V> Values()

Set<Map.Entry<K,V>> EntrySet()

Note: We can only delete elements from set returned by collection-view operation. The set /Collection will be backed by actualmap. We can’t add elements to these sets/collections

Public Interface NavigableMap<K,V> extends SortedMap<K,V>

{

//closest matches

K lowerKey/higherKey/floorKey/ceilingKey(K key)

Map.Entry<K,V> lowerEntry/higherEntry/floorEntry/ceilingEntry(K Key)

NavigableSet<K> descendingKeySet()

NavigableMap<K,V> descendingMap()

Map.Entry<K,V> pollFirstEntry()

Map.Entry<K,V> pollLastEntry()

//range view

NavigabelMap<K,V> headMap(K toKey, Boolean inclusive)

**TreeMap**

It supports methods such as containsKey, get, put, remove with log(n) time complexity.

**private** **static** **void** treeMapDemo() {

System.***out***.println("\n\nInside treeMapDemo ...");

TreeMap<String, Integer> map1 = **new** TreeMap<>();

map1.put("John", 25);

map1.put("Raj", 29);

map1.put("Anita", 23);

System.***out***.println(map1);

System.***out***.println("Iterating using entrySet ...");

Set<Map.Entry<String, Integer>> mappings = map1.entrySet();

**for** (Map.Entry<String, Integer> mapping : mappings) {

System.***out***.println("Name: " + mapping.getKey() + ", Age: " + mapping.getValue());

**if** (mapping.getKey().equals("John"))

mapping.setValue(26);

}

System.***out***.println(map1);

// map1.floorEntry("Raj").setValue(22); - unsupported operation exception. outcome not backed by original treeMap

}

**Java.util.Arrays**

// asList() ~ most commonly used

// List<T> asList(T...)

String[] strArray = **new** String[]{"Raj", "Anita"};

List<String> strings = Arrays.*asList*(strArray); // fixed-size

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

//strings.remove(0); since it is of fixed size, can only set the existing element. Can’t remove or add element

//strings.add("a");

strings.set(0, "john");

System.***out***.println("Updated Array: " + Arrays.*toString*(strArray));

// Creating modifiable ArrayList from an Array

strings = **new** ArrayList(Arrays.*asList*(strArray));

// Showing varargs

strings = Arrays.*asList*("Raj", "Anita");

List<String> fixedList = Arrays.*asList*(**new** String[3]);

// recall from autoboxing that arrays are not auto-boxeable

//List<Integer> fixedList2 = Arrays.asList(new int[2]);

List<**int**[]> fixedList2 = Arrays.*asList*(**new** **int**[2]);

System.***out***.println("fixedList2.size: " + fixedList2.size());

// Sorting: void sort(Object[]) - Uses Merge-sort with natural ordering

// Partially sorted array: far fewer than nlog(n) comparisons

// Almost sorted array: approx. n comparisons, where n is array size

Arrays.*sort*(strArray);

System.***out***.println(Arrays.*toString*(strArray));

// Additional Comments: Well-suited for merging 2 or more sorted arrays

// Concatenate the arrays & sort the resulting array!!

// Sorting: void sort(int[]) - Uses quick sort

**int**[] iArray = {23, 4, 59};

Arrays.*sort*(iArray);

System.***out***.println("Sorted iArray: " + Arrays.*toString*(iArray));

// void sort(T[] a, Comparator<? super T> c)

// Binary Search: int binarySearch(int[], int);

// returns index if element found

// otherwise returns -(insertion point) - 1

// input array must be sorted. Otherwise, behavior is undefined

System.***out***.println("index returned by binary search: " + Arrays.*binarySearch*(**new** **int**[] {4, 23, 59}, 5));

**int**[] newArray = Arrays.*copyOf*(iArray, 6); //copies 4,23,59 and remaining 3 elements are filled with 0

System.***out***.println("newArray: " + Arrays.*toString*(newArray));

**int**[] newArray1 = **new** **int**[6];

System.*arraycopy*(iArray, 0, newArray1, 0, iArray.length);

System.***out***.println("newArray1: " + Arrays.*toString*(newArray1));

Arrays.*fill*(newArray, 13);

System.***out***.println("Fill with 13: " + Arrays.*toString*(newArray));

System.***out***.println("Equals? " + Arrays.*equals*(iArray, newArray));

// Arrays.deepEquals(Object[], Object[]);

// Returns true if arrays are deeply equal to one another.

// Appropriate for nested arrays

//int[][][] deepArray1 = { { {1, 2, 3}, {4, 5, 6} } };

//int[][][] deepArray2 = { { {1, 2, 3}, {4, 5, 6} } };

//int[][] deepArray1 = {{1, 2, 3}};

//int[][] deepArray2 = {{1, 2, 3}};

//int[] deepArray1 = {1, 2, 3}; // Covariance: Does not work as int[] is not a subtype of Object[]

//int[] deepArray2 = {1, 2, 3};

//System.out.println("Deep Array Equals? " + Arrays.deepEquals(deepArray1, deepArray2));

Object[] objArray = **new** **int**[][][] { { {1, 2, 3}}};

**int**[][] ia = (**int**[][])objArray[0];

System.***out***.println(ia[0][2]);

**Note:** deepEquals needs 2 object arrays as input araguments. Int[] is an object..not object array. So,

int[] deepArray1 = {1, 2, 3};

int[] deepArray2 = {1, 2, 3};

Arrays.deepEquals(deepArray1, deepArray2); doesn’t work

**Output**

strings: [Raj, Anita]

Updated Array: [john, Anita]

fixedList2.size: 1

[Anita, john]

Sorted iArray: [4, 23, 59]

index returned by binary search: -2

newArray: [4, 23, 59, 0, 0, 0]

newArray1: [4, 23, 59, 0, 0, 0]

Fill with 13: [13, 13, 13, 13, 13, 13]

Equals? false

3

**Parallelised operations**

If the system has multiple cores, we can make use of multiple cores to achieve parallelism.

parallelSort() is applicable for arrays of large size [at least 2^13].

**Code:**

**private** **static** **void** parallelised() {

// For large arrays on multi-core. Min size atleast 1 >> 13 = 8192

**int**[] iArray = {23, 4, 59};

Arrays.*parallelSort*(iArray);

System.***out***.println("iArray parallel sort: " + Arrays.*toString*(iArray));

IntBinaryOperatorimpl IntBinaryOperatorImpl = **new** IntBinaryOperatorimpl();

Arrays.*parallelPrefix*(iArray, IntBinaryOperatorImpl);

System.***out***.println("Parallel Prefix: " + Arrays.*toString*(iArray));

IntUnaryOperatorimpl intUnaryOperatorImpl = **new** IntUnaryOperatorimpl();

intUnaryOperatorImpl.setArray(iArray);

Arrays.*parallelSetAll*(iArray, intUnaryOperatorImpl);

System.***out***.println("Parallel Set All: " + Arrays.*toString*(iArray));

}

**import** java.util.function.IntBinaryOperator;

**public** **class** IntBinaryOperatorimpl **implements** IntBinaryOperator {

@Override

**public** **int** applyAsInt(**int** left, **int** right) {

// **TODO** Auto-generated method stub

**return** left+right;

}

}

**import** java.util.function.IntUnaryOperator;

**public** **class** IntUnaryOperatorimpl **implements** IntUnaryOperator{

**int** [] arr;

**public** **void** setArray(**int**[] arr) {

**this**.arr=arr;

}

@Override

**public** **int** applyAsInt(**int** operand) {

// **TODO** Auto-generated method stub

**if**(arr!=**null**) {

**return** arr[operand]+5;

}

**else**

**return** operand;

}

}

**Output**

iArray parallel sort: [4, 23, 59]

Parallel Prefix: [4, 27, 86]

Parallel Set All: [9, 32, 91]

**Java.util.Collections**

* **This is “Collections” class**

**Code**

**private** **static** **void** CollectionsDemo() {

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

list.add("Raj");

list.add("John");

list.add("John");

String[] array = {"Anita"};

Collections.*addAll*(list, array);

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

// list.addAll(Arrays.asList(array))

// static <T extends Comparable<? super T>> void sort(List<T> list)

Collections.*sort*(list);

System.***out***.println("Sorted list: " + list);

// <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)

// Needs to be sorted. O(log(n)).

// If not sorted results are undefined. Might perform an inefficient linear search!!

// For Sets: HashSet ~ O(1). TreeSet ~ O(log(n)), i.e., same efficiency as binary search

// Note: List.contains is O(n)

System.***out***.println("Is John There? : " + Collections.*binarySearch*(list, "John"));

// <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)

Collections.*reverse*(list);

System.***out***.println("Reverse list: " + list);

Collections.*swap*(list, 0, 3);

System.***out***.println("After swapping: " + list);

System.***out***.println("# John's: " + Collections.*frequency*(list, "John"));

Collections.*shuffle*(list);

System.***out***.println("Shuffled list: " + list);

System.***out***.println("Max: " + Collections.*max*(list)); // natural ordering

System.***out***.println("Min: " + Collections.*min*(list));

// Singleton ~ <T> Set<T> singleton(T o)

// Returns an immutable set containing only the specified object

// Others: <T> List<T> singletonList(T o) & <K,V> Map<K,V> singletonMap(K key, V value)

list.removeAll(Collections.*singleton*("John"));

// Unmodifiable View ~ to provide clients with read-only access to internal collections

Collection<String> unmodifiable = Collections.*unmodifiableCollection*(list);

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

System.***out***.println("Is Anita there?: " + unmodifiable.contains("Anita"));

//unmodifiable.add("John");

list.add("John");

System.***out***.println("unmodifiable 2: " + unmodifiable);

// Check out other methods in the API

// ~ Methods starting with keyword empty like emptyList will be briefly discussed in

// a subsequent lesson when discussing an item from Effective Java

// ~ Check back on methods starting with name "checked" as they are generics related

// ~ Synchronized methods will be reviewed in Concurrency chapter

}

Output

Is John There? : 1

Reverse list: [Raj, John, John, Anita]

After swapping: [Anita, John, John, Raj]

# John's: 2

Shuffled list: [Anita, John, John, Raj]

Max: Raj

Min: Anita

unmodifiable: [Anita, Raj]

Is Anita there?: true

unmodifiable 2: [Anita, Raj, John]

**Effective java item**

If the method has to return array/collection and it doesn’t have anything to return, it is better to return an empty array/collection instead of null.

* If null is returned by the method, client will have to write some extra code for null check
* Isn’t it expensive to return emty array/collection every time the search fails?

Can return the same empty array/empty immutable Collection

**Idiom for returning Array from collection**

Private static String[] EMPTY\_ARRAYnew String[0];

Public String[] getBookTitleByAuthor(String author){

List<String> titles=dao.getBookTitlesbyauthor(author);

Return titles.toArray(EMPTY\_ARRAY);

}

//returns input array if it is large enough

**Idiom for returning same immutable collection**

Public String[] getBookTitleByAuthor(String author){

List<String> titles=dao.getBookTitlesbyauthor(author);

If(titles.isEmpty()){

Return Collections.emptyList();

}

return titles;

}

Conclusion

List - To be used when sequence/positioning is impotant.

ArrayList- resizable array implementation of List interface.

* Fast random access O(1)
* Inserting or deleting elements at the end is fast O(1)

LinkedList: Doubly Linked List implementation of List and Deque interface

-> Can be used to frequently add/ delete elements during iteration.

Queue

* For fast head or tail manipulations.

ArrayDeque: Array implemtation of Deque interface.

* Supports LIFO or FIFO operations in O(1).
* It is always better to use ArrayDeque instead of LinkedList when LinkedList has to be used as Queue.

Set

* Used when uniqueness and fast lookups matter.

HashSet

* HashTable implementation of Set interface.
* Very fast lookup, insertion, deletion O(1).
* Always override the hashCode method when we override the equals method.

LinkedHashSet

* Insertion order is preserved + fast look up O(1).

TreeSet

* Implementation of NavigableSet interface
* Sorting + contains, add , remove operations in O(log(n)).

HashSet -------- HashMap

LinkedHashSet------LinkedHashMap [L R U cache]

TreeSet --------Tree Map

[Sets internally use Maps]

* HashSet and TreeSet extends AbstractSet
* HashMap and TreeMap extends AbstractMap
* AbstractSet extends AbstractCollection [skeletal implementation of Collection interface]

Arrays

* copyOf, asList, binarySearch, sort, parallelSort, equals
* asList – returns a fixed sized list. So, we can’t perform add/remove operations on this list.
* To convert an array into list

New HashSet(new Arrays.asList(array));

Collections

* addAll, sort, binarySearch, unmodifiableCollection, swap, reverse, shuffle

Note: better to return empty arrays or Collections instead of nulls

**Introduction to Generics**

Polymorphism -> generalization

Eg: public class Store{

Object a;

void set(Object a){

a=this.a;

}

Object get( ){

return a;

}

}

**Issues due to this piece of code**

John:

Store store= new store( new Date(someDate)); //java.sql.Date

Date date= (Date)store.get();

Anitha:

Store.set(new Date()); //java.util.Date

John:

Date date = (Date)store.get(); //ClassCastException

* The problem with this approach is that the type is way too generic, needs to down casted, leads to runtime exception.

Code:

class Store<T>{

private T a;

public void set(T a ){

this.a=a;

}

public T get(){

return a;

}

}

**John:**

Store<Date> store= new Store<>();

store.set( new date(sometime)); //java.sql.Date

Date date=store.get(); //no type casting

**Anitha:**

store.set(new Date()); //compile time error

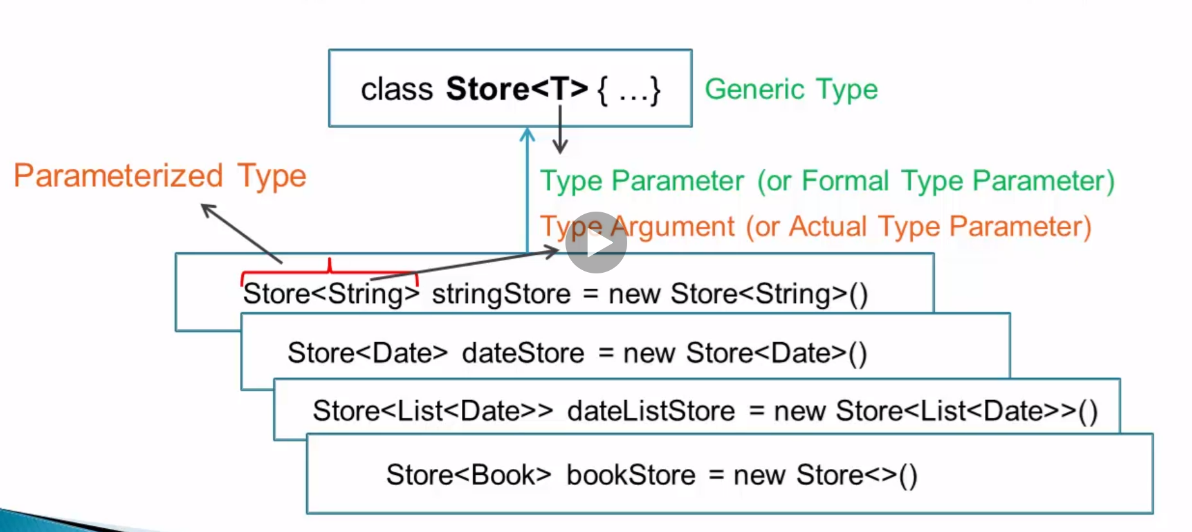
**Raj:**

Store<Book> store=new Store<>();

Store.set(new Book());

//type safety at compile time

* Generics can be used as type of parameters, local variables, return type



Store<Book> bookstore=new Store<>();

//instead of Store<Book> bookstore=new Store<Book>();

[“<>” is called “diamond notation”. Works for only java 7 onwards].

**Type parameter naming conventions**

**E –** element of Collection

**K**- key, **V**- Value

**N**- Number

**T**- type( usually non Collection type)

**S,U,V** – 2nd, 3rd, 4th elements

**Sub typing of Generic types**

public class Store<T> implements Container<T>{

private void set(T a){

this.a=a;

}

private T get(){

return a;  
}}

<T> - parameterised type

Public interface container<T>

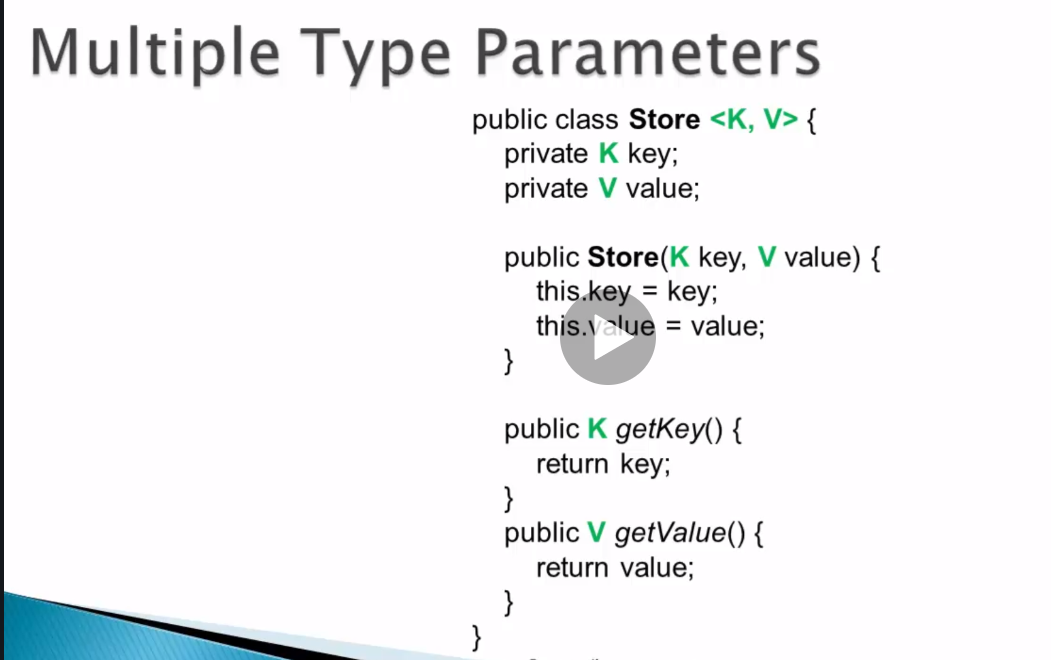
{

private void set (T a);

private T get();

}

Container<T> store= new Store<>();



* Type argument can’t be primitive

Eg: Store<int> store =new Store<>();

* Type parameter can’t be used in static context

Eg:

public class Device<T>{

private static T DeviceType;

}

Device<phone> device=new Device<>();

Device<pager> device = new Device<>();

[If code is written like this..what would be the type of the static variable? Hence, type parameter is not allowed to be used in static context].

* Type parameter can’t be used in static method or static initializer too.

Demo:

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** GenericsDemo {

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

Store<String> stringStore=**new** Store<>();

stringStore.set("java");

//stringStore.set(1); - compile time error

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

Store<List<Integer>> listStore=**new** Store<>();

listStore.set(Arrays.*asList*(1,2,3));

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

List<Number> list1=**new** ArrayList<>();

list1.add(**new** ~~Integer~~(1));

list1.add(**new** ~~Double~~(22.1));

System.***out***.println(list1);

List[] listArray=**new** List[2];

listArray[0]=**new** ArrayList<>();

listArray[1]=**new** LinkedList<>();

}

}

**class** Store<T> **implements** Container<T>{

**private** T a;

**public** **void** set(T a) {

**this**.a=a;

}

**public** T get() {

**return** a;

}

}

**Output:**

java

[1, 2, 3]

[1, 22.1]

* Generics is compile time concept
* It uses a concept of type erasure

i.e. internally …<T> will be stripped off, T will be replaced by “Object” or super type of class

public class Store implements Container{

Object a;

void set(Object a){

a=this.a;

}

Object get( ){

return a;

}

}

When we call the getter method

Eg: Store<String> stringStore=**new** Store<>();

stringStore.set("java");

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

The returned type will be object. It will be internally type casted to String.

* If we change the type of Container, we will have to add unimplemented methods in store class

Eg: Class Store<T> implements Container<String>

{

**private** T a;

**public** **void** set(T a) {

**this**.a=a;

}

**public** T get() {

**return** a;

}

public void set(String a)

{

this.a=a;

}

public String get(){

return a;

}

}

**Bounded type parameter**

* Parameter with one or two bounds.

Eg: class GenericsDemo<T extends List>

GenericsDemo<List> test = new GenericsDemo<>();

GenericsDemo<ArrayList> test = new GenericsDemo<>();

GenericsDemo<LinkedList> test = new GenericsDemo<>();

**Advantage:** Can access methods defined by bounds

Eg 1: class GenericsDemo<T>{

void go( T list){

int i=list.size(); //compiler error since compiler is not sure whether T has the method size()

}

}

class GenericsDemo<T extends List>{

void go( T list){

int i=list.size(); // no compiler error }

}

Type erasure:

->If there is no bounded type parameter –“T” would be replaced by Object in byte code

->If there is bounded type parameter –“T” would be replaced by he bound in byte code

**Valid bounds**

1] class

2] interface

3] enum

4] parametrized type – eg: class GenericsDemo<T extends Collection<T>>

**Invalid bounds**

1] primitives

2] arrays

**Specifics**

1] Type argument must be sub type of all bounds

class GenericsDemo<T extends List & Serializable>

GenericsDemo<List> test= new GenericsDemo<>(); //compiler error – List is not sub type of serializable

GenericsDemo<ArrayList> test= new GenericsDemo<>(); //no compiler error – ArrayList is sub type of both list and Serializable

* If class is one of the bounds, it must be first.
* If first bound is class, remaining must be interfaces.
* If the bound is final class or enum, the type argument is the bound itself.

**code**

import java.util.ArrayList;

import java.util.Arrays;

import java.util.LinkedList;

import java.util.List;

public class GenericsDemo <T extends List> {

public static void main(String args[]) {

GenericsDemo<ArrayList> test1=new GenericsDemo<>();//no compiler error

// GenericsDemo<Collection> test2=new GenericsDemo<>(); - compiler error - Collection is not subtype of List

}

public void go(T list) {

list.add(0, new Object()); //we get compiler error if T doesn't extend List interface

}

}

**Raw Type**

* Raw type is Generic Type without type argument

Eg: List list=new ArrayList();

void go(List list)

* Parameterised type is generic type without type argument.
* Consequence of raw type is that compile time safety is lost

Eg 1:



All the 4 classes are responsible for fetching the prize of a given item from a different source. These classes are a part of data integration component which integrates the prices from different sources.

Code snippet:

Public List<Double> fetchData(int ISBN){

List<Double> prices=new ArrayListy<>();

(new BNIntegrator()).getPrice(ISBN, prices);

(new HalfIntegrator()).getPrice(ISBN, prices);

Double price= prices.get(2); // ClassCastException

}

public class HalfIntegrator{

public void getPrice(int ISBN, List prices){ //no parameterised type

int price= HalfAPIParser.getprice(ISBN);

prices.add(price); //int is added …because the List is not parameterised type here

}

**Exceptions for using raw types**

1]Class literals

List.class; //correct

List<String>.class //incorrect

2]Instance of

if( o instanceOf List) //correct

if( o instanceOf List<String>) //incorrect

**Demo code:**

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** GenericsDemo <T **extends** List> {

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

//rawTypeDemo(); - this will give classCastException

List<String> list2=Arrays.*asList*("a","b","c");

List<String> list3=Arrays.*asList*("d","b","c");

System.***out***.println(*getNumcommonElements*(list2,list3));

}

**public** **static** **void** rawTypeDemo() {

**int** ISBN=122324;

List<Double> prices=**new** ArrayList<>();

HalfIntegrator.*getPrice*(ISBN,prices);

Double price=prices.get(0);

}

**public** **static** **int** getNumcommonElements(List list1,List list2) { //programmers are generally tempted to use raw types when we don't know the actual type/ don't care about actual type

**int** count=0;

**for**(Object element:list1) {

**if**(list2.contains(element)) {

count++;

}

}

**return** count;

}

}

**import** java.util.List;

**public** **class** HalfIntegrator {

**public** **static** **void** getPrice(**int** iSBN, List prices) {

prices.add(45);

}

}

**Unbounded wildcard**

class Store<T>{…}

void go(Store<?> someStore){…} //unbounded wildcard

* It can only be used as type argument and not as type parameter.

**Eg:**

class Store<?,?>{

? a; // does ? refer to 1st or 2nd parameter?

}

Common usage:

void go(Store<?> someStore)

Not common usage:

List<?> list=new ArrayList<>();

**Why not Store<Object> instead of Store<?> ?**

Due to a aproperty called invariance, we can assign only Store<Object> to Store<Object>

Eg: Store<Object> store=new Store<Object>();

But, if we use wildcard…

Store<?> store=new Store<String>();

Store<?> store=new Store<Integer>();

* Can’t invoke methods that use class level type parameters with any arguments except null.

Eg:

**public** **static** **int** getNumcommonElements(List<?> list1,List<?> list2) {

**int** count=0;

**for**(Object element:list1) {

**if**(list2.contains(element)) { //no error

count++;

}

}

list2.add(45); //compilation error

**return** count;

}

Example demo

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** GenericsDemo <T **extends** List> {

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

Container<String> someStore=**new** Store<>();

someStore.set("Mahi");

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

Container<?>store1= someStore;

Object obj=store1.get();

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

/\* List<Integer> intList1 = Arrays.asList(1, 2);

List<Integer> intList2 = Arrays.asList(3, 4);

invalidAggregate(intList1, intList2, new ArrayList());\*/ - compiler error

}

public static void invalidAggregate(List<?> l1, List<?> l2, List<?> l3) {

//l3.addAll(null); // null ok

//l3.addAll(l2);

}

}

**class** Store<T> **implements** Container<T>{

**private** T a;

**public** **void** set(T a) {

**this**.a=a;

}

**public** T get() {

**return** a;

}

}

**Output**

Mahi

Mahi

**Invariance**

* Let Book be sub type of Bookmark. This doesn’t imply that List<Book> is sub type of List<Bookmark>. List<Book> can’t be assigned to List<Bookmark>

If at all this was possible: eg:

void go(ArrayList<Bookmark> items){

items.add(new Movie()); //would have caused class cast exception;

}

go(new ArrayList<Book> items);

* Can’t assign List<Book> to List<Bookmark>

But, can assign ArrayList<Bookmark> to List<Bookmark>

Arrays are covariant

Book -> Bookmark

Book[] -> Bookmark[]

Eg: Bookmark[] items=new Book[1];

* void go(Bookmark[] items){

items[0]=new Movie(); //ArrayStore exception

}

go(new Book[1]);

item 25: prefer List over Arrays [because, Arrays ensure type safety at run time whereas, List ensures type safety at compile time].

Demo:

**public** **class** GenericsDemo <T **extends** List> {

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

//go1(new ArrayList<Integer>); - compile time error

*go2*(**new** Integer[1]);

*}*

**public** **static** **void** go1(List<Number> list) {

}

**public** **static** **void** go2(Number[] num) {

num[0]=23.5; //run time exception

}

Output:

Exception in thread "main" java.lang.ArrayStoreException: java.lang.Double

at GenericsDemo.go2(GenericsDemo.java:56)

at GenericsDemo.main(GenericsDemo.java:43)

**Generic methods**

* Methods and constructors can have their own type parameter.

Syntax: <T1,T2,…> returnType methodName(T1 p1, T2 p2,…)

* We know that if a method is static, it has no access to enclosing class’ type parameter.
* Static utility methods are good candidates of generic methods

Eg: Collections class

1] <T> T[] toArray(T[] a) //java.util.Collection

2] static <T> boolean replaceAll(List<T> list, T oldValue, T newValue)

3] class GenericsDemo<T>{

<T> void go(T obj){ //method level type parameter overrides class level type parameter

}

4]class GenericsDemo <E>{

<T> void go(T obj, E obj){ //uses both method level type parameter and class level type parameter. We will get compilation error if method is static

}

5]Method with bounded type parameter

<T extends List & Serializable> void go(T obj)

6]Method level type parameter extending class level type parameter

class Store<E>{

<T extends E> void go(T obj)

}

7]First type parameter extending second type parameter

<T1, T2 extends T1> void go(T1 obj, T2 obj)

Note: Generic methods can exist whether or not the enclosing type is generic

**Method invocation**

Eg 1: <T> T void go(T object)[

return object;

}

go(1.1); //type argument would be automatically inferred by compiler as Double

go(“java”); //type argument would be automatically inferred by compiler as String

eg 2:<T> List <T> emptyList( );

List<String> list= Collections.emptyList(); //type argument is inferred as String

Eg 3: <T> void go(T obj){…}

Double doub = go(“java”); //compiler error

Eg 4: <T> T go(T a, T b) {…}

Serializable s= go(“d”, new ArrayList<String>());

//type inferred will be the most specific, common super type. Here, both String and ArrayList implement Serializable

Explicit type argument specification

class GenericsDemo{

<T> void go(T obj){..}}

GenericsDemo gd=new GenericsDemo();

gd.<Double>go(1.1);

->If method is in the same class->

This.<Double>go(1.1);

* For method of super class, we would use “super” keyword
* Static method,

GenericsDemo.<Double>go(1.1);

Demo code:

**import** java.io.Serializable;

**import** java.util.AbstractCollection;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.Collection;

**import** java.util.HashSet;

**import** java.util.List;

**public** **class** Generics<T> {

**public**<E **extends** T> Generics(E object) {

}

**public** **static** **void** GenericsDemo() {

*typeArgumentInference*(22.1); //Type argument is inferred as double. Can check this by hovering over it

*typeArgumentInference*("Java");

//Double d=typeArgumentInference1("Java"); - type mismatch

Integer[] a=**new** Integer[100];

Collection<Integer> cs=**new** ArrayList<>();//correct

//Collection<String> cs=new ArrayList<>(); -compilation error

Collection<Number> cs1=**new** ArrayList<>();//correct

*arrayToCollection*(a,cs);

String str=*typeArgInferenceFromReturnType*();//correct

// Integer in=typeArgInferenceFromReturnType(); //class cast exception since the type inferred would be Integer and String is returned

*targetTypeInvoker*(*typeArgInferenceFromReturnType2*()); //type inferred is List<String>

*targetTypeInvoker*(**new** ArrayList<>()); //type inferred should have been List<String>. Bug in Eclipse

*targetTypeInvoker2*(*typeArgInferenceFromReturnType2*());//return type is inferred as List<Object> for both methods

List<String> list=*targetTypeInvoker2*(*typeArgInferenceFromReturnType2*()); //return type is inferred as List<String> for both methods

*targetTypeInvoker2*(**new** ArrayList<>());//return type is inferred as <Object> for both methods

List<String> list1=*targetTypeInvoker2*(*typeArgInferenceFromReturnType2*()); //return type is inferred as List<String> for both methods

Serializable s=*targetTypeinvoker3*("", **new** ArrayList<>()); //Serializable is the inferred type

AbstractCollection abs=*targetTypeinvoker3*(**new** HashSet<>(), **new** ArrayList<>());

Generics.<String>*uselessGenerics*(); //useless because the explicitly specified type argument is not used anywhere in the client code

**new** Generics<Number>(22.1); //T is Number, E=Double

**new** Generics<>(22.1);//T,E are double

**new** <Double>Generics<Number>(22.1);

Generics<Number> gd=**new** Generics<>(22.1); //<Double>Generics.generics<Number>(Double obj) due to invariance

List <Integer> list2=Arrays.*asList*(1,2);

List <Integer> list3=Arrays.*asList*(1,3);

List<Integer> list4=**new** ArrayList<>();

*aggregate*(list2,list3,list4);

}

**private** **static** <E>**void** aggregate(List<E> list2, List<E> list3, List<E> list4) {

list4.addAll(list2);

list4.addAll(list3);

}

**private** **static** <T> **void** uselessGenerics() {

T t=(T)**new** Integer[2];

}

**private** **static** <T> T targetTypeinvoker3(T a, T b) {

**return** (T)a;

}

**private** **static** <T> List<T> targetTypeInvoker2(List<T> list) {

**return** (List<T>)list;

}

**private** **static** **void** targetTypeInvoker(List<String> str) {

**for**(String a:str)

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

}

**private** **static** <T> List<T> typeArgInferenceFromReturnType2() {

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

list.add("abc");

**return** (List<T>)list;

}

**private** **static** <T> T typeArgInferenceFromReturnType() {

**return** (T) "abc";

}

**private** **static** <T> **void** arrayToCollection(T[] a, Collection<T> cs) {

**for**( T o:a)

cs.add(o);

}

**private** **static** <T> **void** typeArgumentInference1(T object) {

System.***out***.println(object.getClass().getName());

}

**private** **static** <T> T typeArgumentInference(T object) {

System.***out***.println(object.getClass().getName());

**return** object;

}

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

*GenericsDemo*();

}

}

Output:

java.lang.Double

java.lang.String

[abc]

**Bounded Wildcards**

**->**Generics are invariant. [If Book is a sub type of Bookmark, List<Book> is not a sub type of List<Bookmark>

Eg: void go(List<Bookmark> items)

{

items.add(new Movie());

}

Go(new ArrayList<Book>()); //error

Let us consider a method: void display(List<Bookmark>) ->It just displays the Bookmark [Here type safety is not important]

But due to generics, we can’t pass ArrayList<Book> or ArrayList<Movie> as an argument

Soln 1: void display(List<Bookmark> items)

void display(List<Book> books) [having separate methods]

This solution is not possible due to type erasure. It will create duplicate code. Method over loading is not possible with generics.

1]Can use upper bounded wild card-

void display(List<? Extends Bookmark> items)

2]Generic method with bounded type parameter

<T extends Bookmark> void display(List<T> items)

**Another restriction due to invariance**

void aggregate(List<Bookmark> items){

items.add(new Bookmark());

items.add(new Movie());

items.add(new Book());

}

aggregate(new ArrayList<Bookmark>()); //valid

aggregate(new Arraylist<Object>()); //invalid – can’t even pass ArrayList of super type of Bookmark [not only sub type of Bookmark]

solution is **lower bounded wild card**

void aggregate(? super Bookmark){…}

demo code:

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Generics {

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

*invarianceDemo*();

}

**public** **static** <T **extends** Number> **void** Invariance(List<T> list) {

//list.add(new Double(23.1)); - not of type T

/\* T t=(T)new Double(23.1);\*/

// list.add(t); // leads to class cast exception

}

**public** **static** <T **extends** Number> **void** Invariance(List<T> list, T num) {

list.add(num);

}

**public** **static** **void** invarianceDemo() {

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

*Invariance*(in);//no error

*Invariance*(**new** ArrayList<Number>()); //no error

//Invariance(new ArrayList<String>()); - compile time error

// Integer ins=in.get(0);

*Invariance*(in,21);

Integer ins=in.get(0);

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

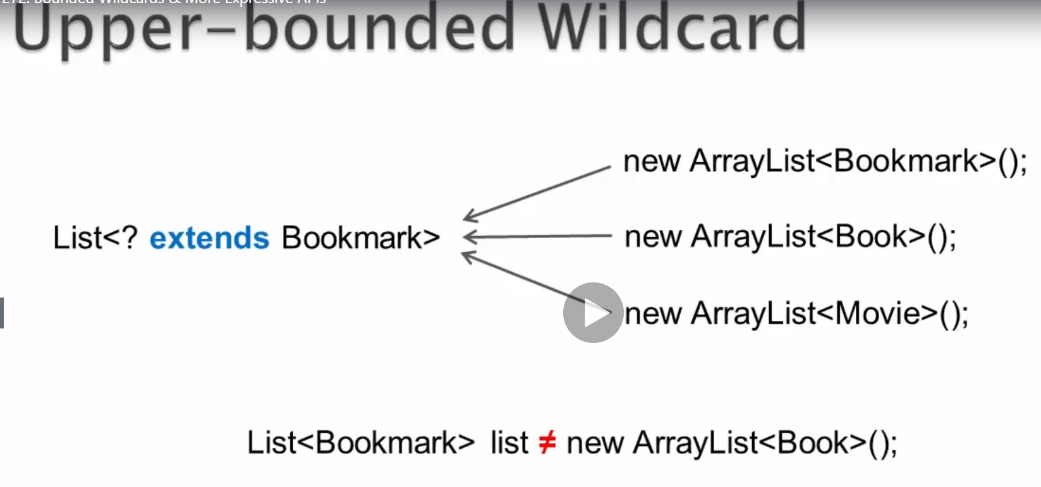
}

}

Output

21

**Bounded wild cards**



**Note:** Due to invariance, we can’t assign ArrayList<Book> to List<Bookmark>

List<Bookmark> list=new ArrayList<Book>(); //error

**Solution 1 for invariance:**

void display(List<? extends Bookmark> list)

{

for(Bookmark bookmark:list){

System.out.println(bookmark);

}}

**Solution 2 for invariance**

<T extends Bookmark> void display(List<T> list)

{

for(Bookmark bookmark:list){

System.out.println(bookmark);

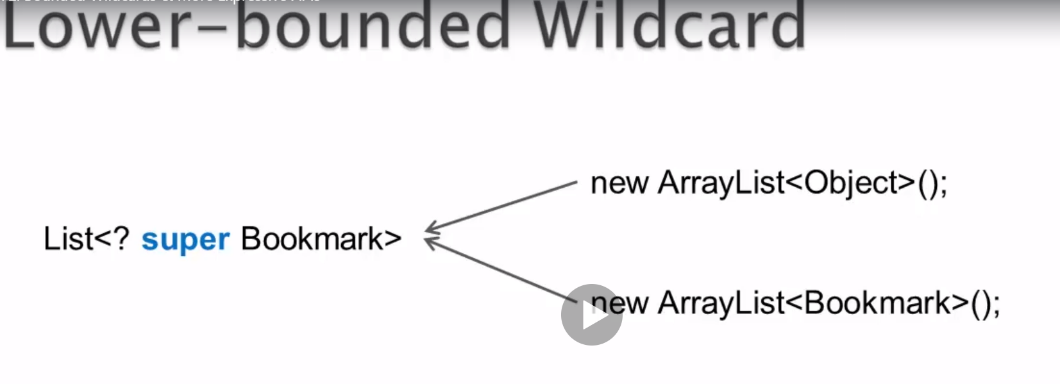
}}

* Unbounded wild card <?> means <? extends Object>

**Compiler type safety restriction**

->We cannot invoke any method that uses class level type parameter with argument other than null.

Eg: If type of wildcard is ArrayList, we can’t invoke add method which uses class level type parameter



List<? super Bookmark> = new ArrayList<Book>(); //incorrect

Type safety restriction for lower bounded wild cards

* We can invoke the methods with class level type parameter only if the arguments are of Lower bounded type or one of it’s sub types

Eg:

void aggregate(List<? super Bookmark> list){

list.add(new Bookmark(..));

list.add(new Book(..));

list.add(new Movie(..));

}

aggregate(new ArrayList<Bookmark>());

aggregate(new ArrayList<Object>());

* void aggregate(List<? super Bookmark> list){

Bookmark b = list.get(0); //invalid. If List is of the type object, we get ClassCastException

Object o = list.get(0); //valid

* Lower bound is only for wildcard ..

<T super LowerBound> is illegal

* ? can have only a single upper or lower bound

<? extends bound1 & bound2> //illegal

* Item 28: Use Bounded wildcard to increase API’s flexibility
* Use upper bound wild card, when parameterised type produces data

<? extends Bound>

* Use lower bound wildcard when parametrised type consumes data

<? super Bound>

* If parameterised type produces data and we don’t know / don’t care about type argument -> we can use unbounded wild card
* If parameterised type produces and consumes data, we have to use exact match

**Methods in Collection class**

<T> boolean addAll(Collection<? super T> collection, T…elements) //consumer

<T> void copy(List<? super T> dest, List<? extends T>) //dest is the consumer and src is the producer

boolean disjoint(Collection<?> col1, Collection<?> col2) //don’t care about type

replaceAll(List<T> list, T oldVal, T newVal){

for(int i=0;i<size;i++){

if(oldVal.equals(list.get(i))){

//acting as producer

List.set(i, newVal); //consumer

} //since both producer and consumer, we need the exact match

**Note:**

* Use generic methods when the same type parameter is used multiple times in method declaration or return type

**eg:** <T> Boolean replaceAll(List<T> list, T oldVal, T newVal)

demo code:

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.Collections;

**import** java.util.List;

**public** **class** Generics {

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

// invarianceDemo();

*WildCardDemo*();

}

**public** **static** **void** WildCardDemo() {

List<Integer> list1= Arrays.*asList*(1,4,5);

*display*(list1);

List<Double> list2=Arrays.*asList*(10.2,20.3,30.0);

*display*(list2);

List<Number> list3=**new** ArrayList<>();

*aggregateWithConsumer*(list1,list2,list3);

Collections.*addAll*(**new** ArrayList<Object>(), 11,23,44); //type inferred will be the most specific type..i.e Integer

System.***out***.println(list3);

Collections.*copy*(list3, list1); //type inferred is Integer - copies element at 0th pos of list1 to 0th pos of list3 and so on...

System.***out***.println(Collections.*disjoint*(list2, list1)); //unbounded wild card

System.***out***.println(list3);

*replaceAll*(list3,5,10);

System.***out***.println(list3);

ArrayList<Number> arrList=**new** ArrayList<>(list1); // this is possible because of ArrayList constructor - ArrayList(Collection<? extends E> c) - Here E is "Number"

}

**public** **static** **void** display(List<? **extends** Number> list) {

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

System.***out***.println(element);//producer

}

}

**public** **static** **void** adder(List<? **super** Number> list) {

list.add(10);

list.add(23.3);

**for**(Object e:list) { //Not recommended to use as producer when super is used

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

}

}

**public** **static** <T **extends** Number> **void** display1(List<T> list){ //solution2- alternative to wild card. preferred when type paramter is used multiple times in method arguments or return type

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

System.***out***.println(element);//producer

}

}

**public** **static** <E> **void** aggregateWithConsumer(List<? **extends** E> list1, List<? **extends** E> list2, List<? **super** E> list3) {

list3.addAll(list1);

list3.addAll(list2);

System.***out***.println(list3);

}

**public** **static** <E1 **extends** E3,E2 **extends** E3,E3> **void** aggergateWithConsumer2(List<E1> list1, List<E2> list2, List<E3> list3) { //alternative method

list3.addAll(list1);

list3.addAll(list2);

}

**public** **static** <T> **void** replaceAll(List<T> list, T oldVal, T newVal) {

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

**if**(oldVal.equals(list.get(i))) {

list.set(i, newVal);

}

}

}

}

**Output**

4

5

10.2

20.3

30.0

[1, 4, 5, 10.2, 20.3, 30.0]

[1, 4, 5, 10.2, 20.3, 30.0]

true

[1, 4, 5, 10.2, 20.3, 30.0]

[1, 4, 10, 10.2, 20.3, 30.0]

**Generics Restrictions**

* Type arguments can’t be primitive

Eg: Store<int> intStore = new Store<int>();

* Class level type parameter can’t be used in static context

i.e. it can’t be used as a type for static variable, can’t be used in static methods or static initializer

* Can’t overload methods that have same signature after type erasure

Eg: void display( List<String> list)

void display(List<Integer> list)

* Generics and Arrays don’t mix well

Eg: new ArrayList<String>[1] //Generic Array creation error at compile time

new E[1]

**If Generics Arrays were possible**

**Eg:** Object[] stringList=new ArrayList<String>2];

stringList[0]= new ArrayList<String>();

StringList[1] =new ArrayList<Integer>(); //This is possible since the reference type is array of Object

We would expect “ArraysStoreException” at run time. That would not happen due to type erasure. Hence type safety is broken

* Arrays are reified- it means that array knows the type of element to be stored at run time. If we store a wrong type, we would get ArrayStoreException
* Generics are non reified
* Exceptions and error types can’t be generic

Eg: class MyExceptionClass<T> extends Exception{..} //any class extending Throwable or it’s sub types can’t have generic type

If this was possible, we would be having something like this-

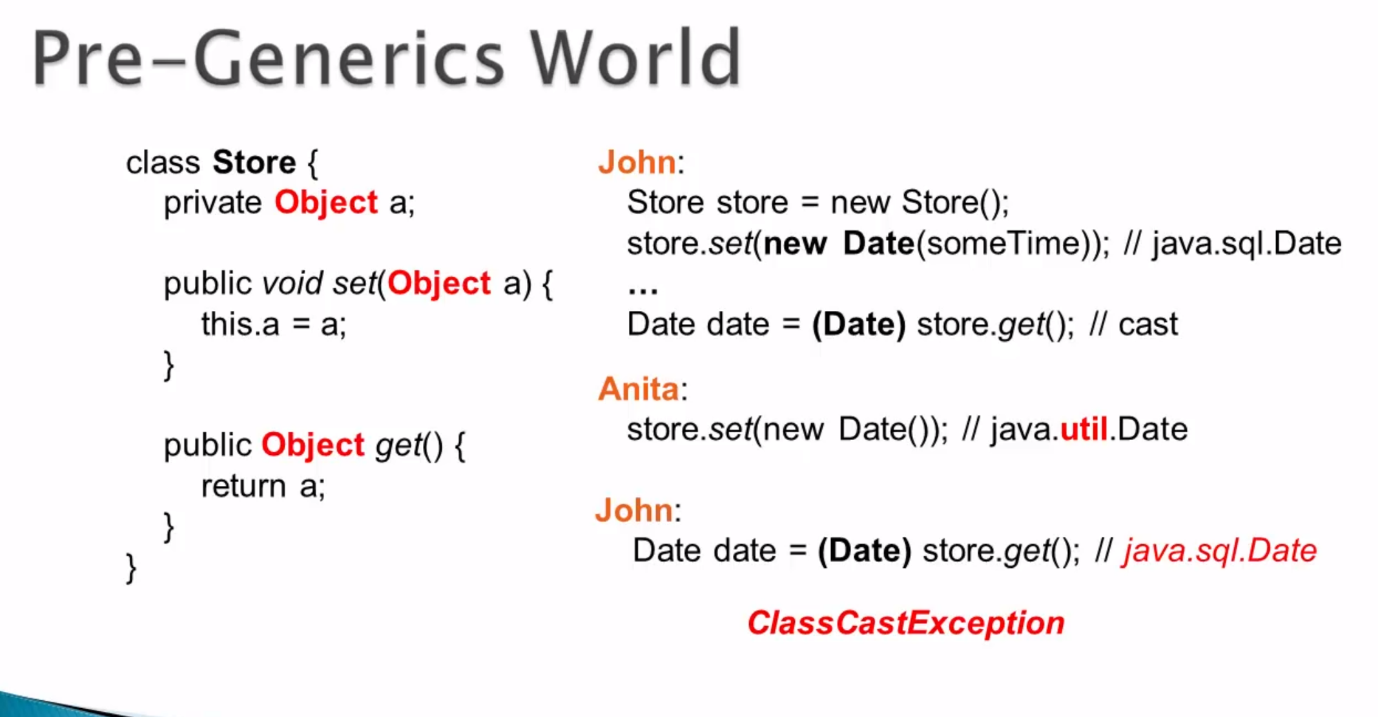
try{

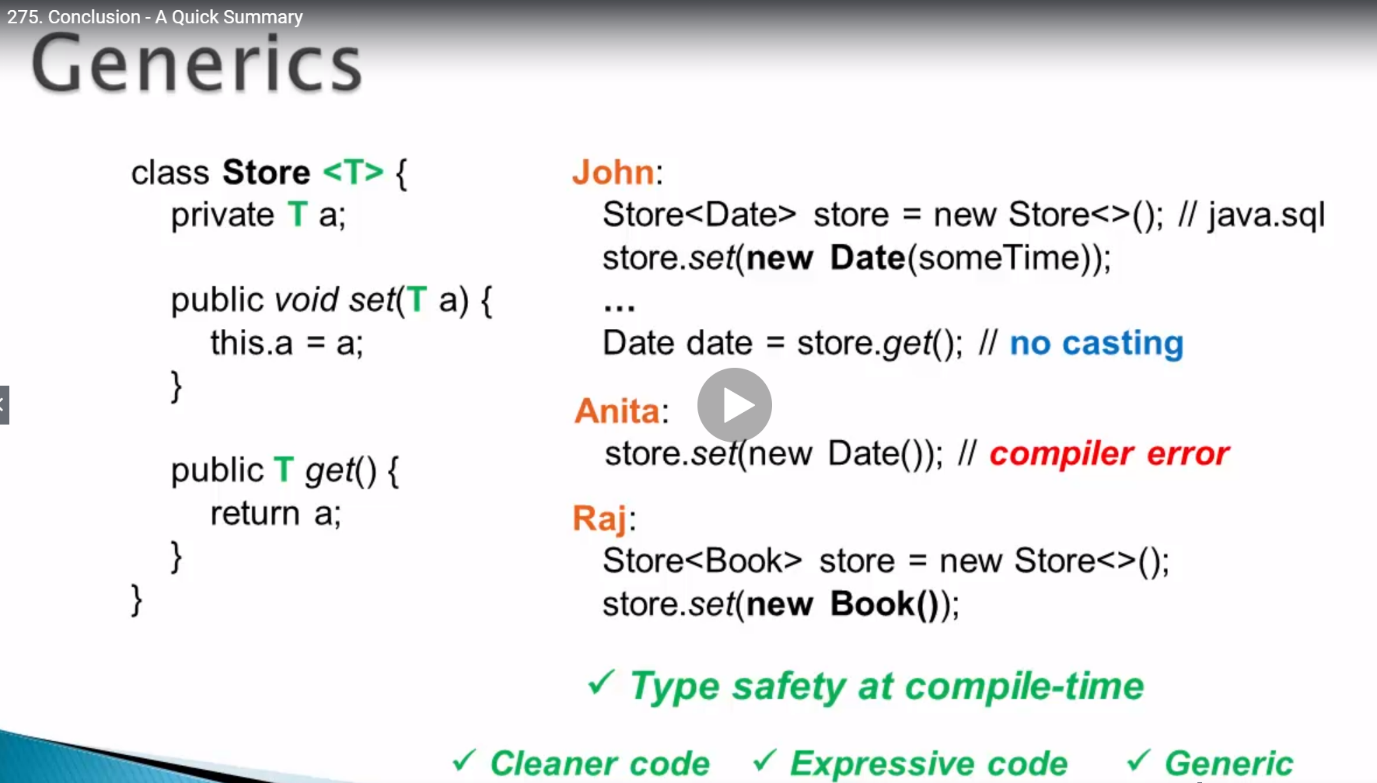
}catch(MyExceptionClass<String> e){..}

catch(MyExceptionClass<Integer> e){…} //but due to type erasure, both the catch blocks would be considered as duplicates

**Summary**

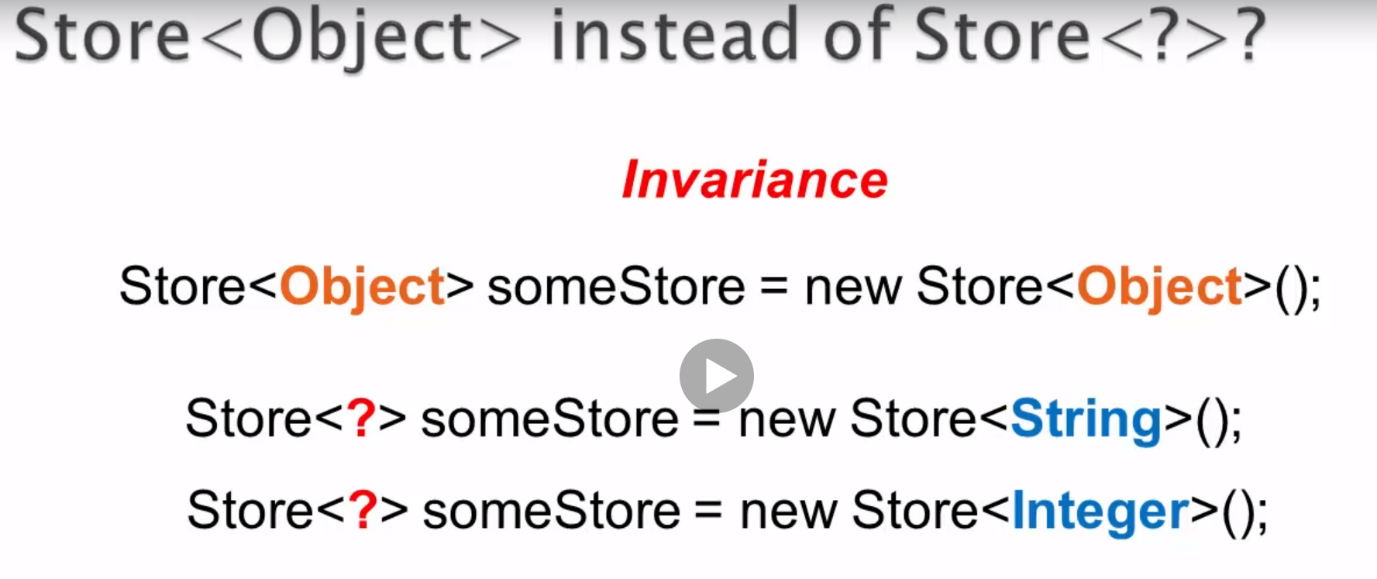
main benefit- compile time type safety

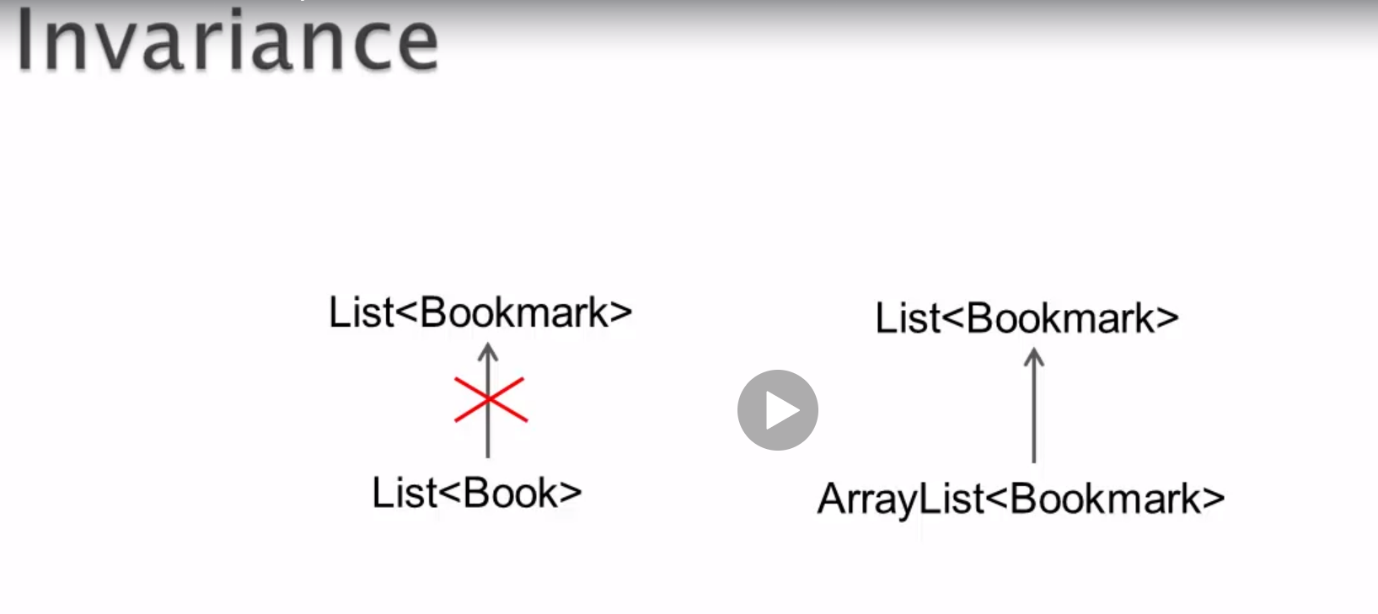




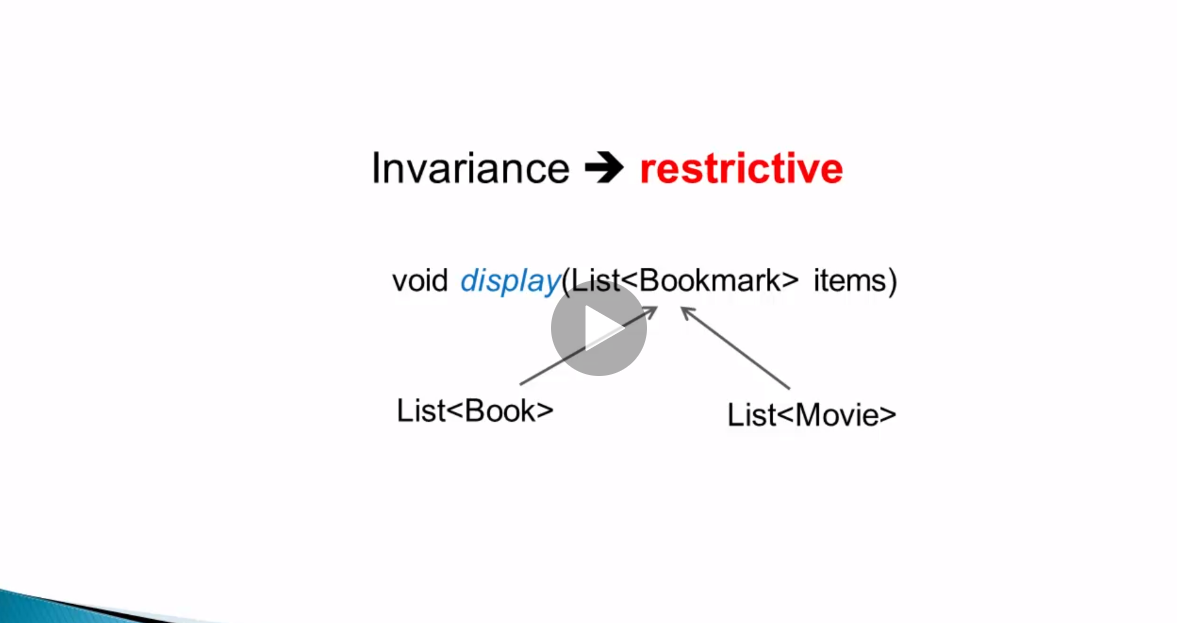
**Unbounded wildcard**

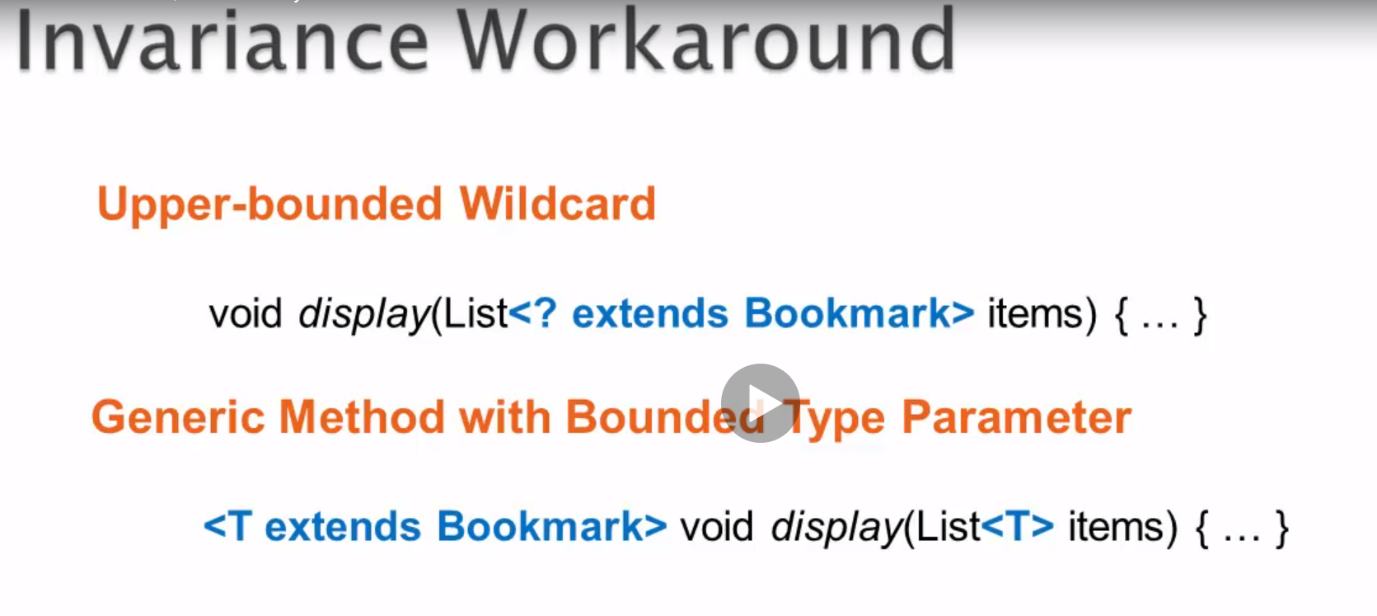
* void go(List<?> list)
* It can be used as type argument , not as type parameter

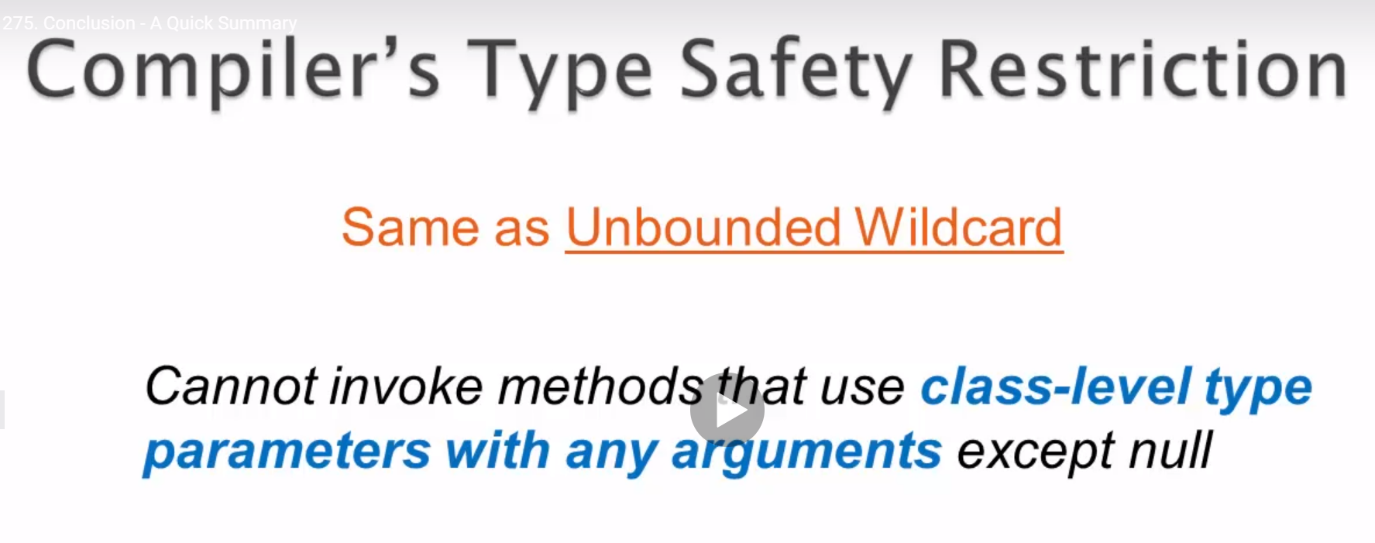


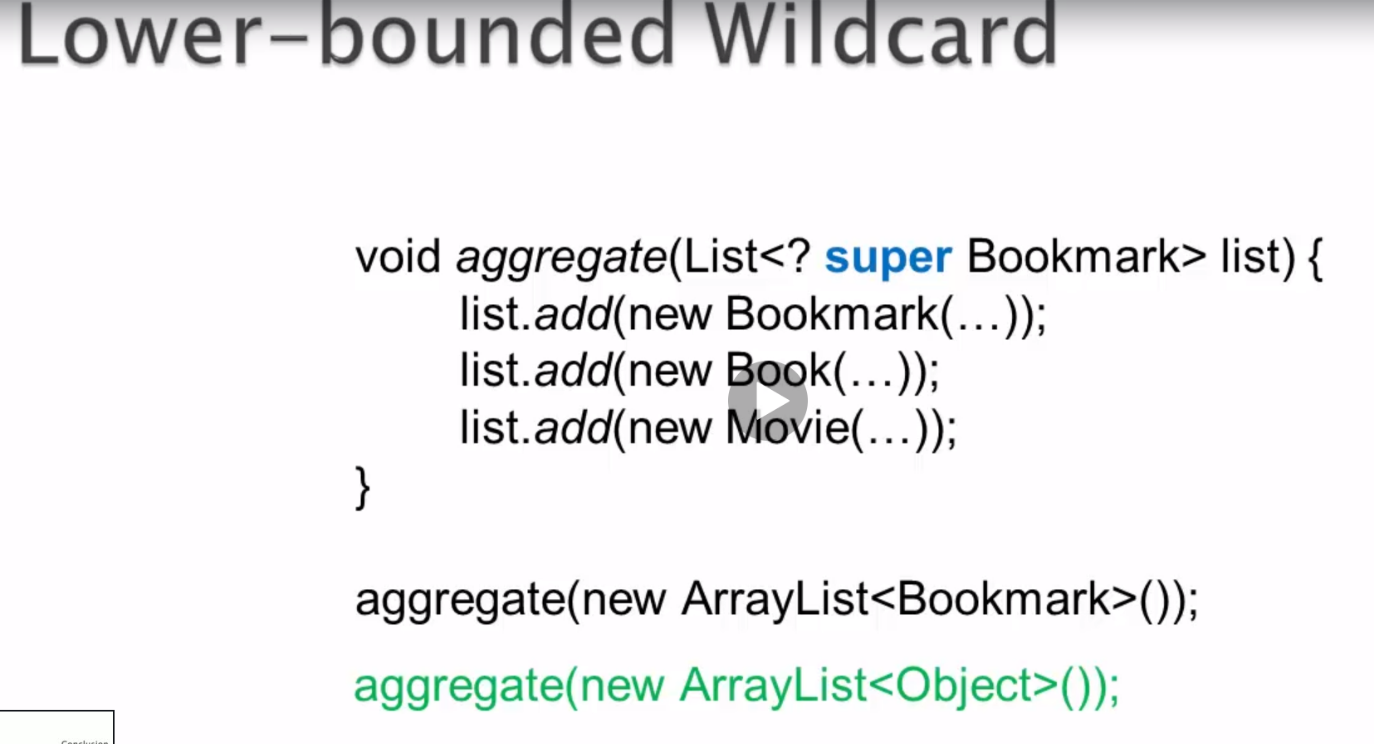


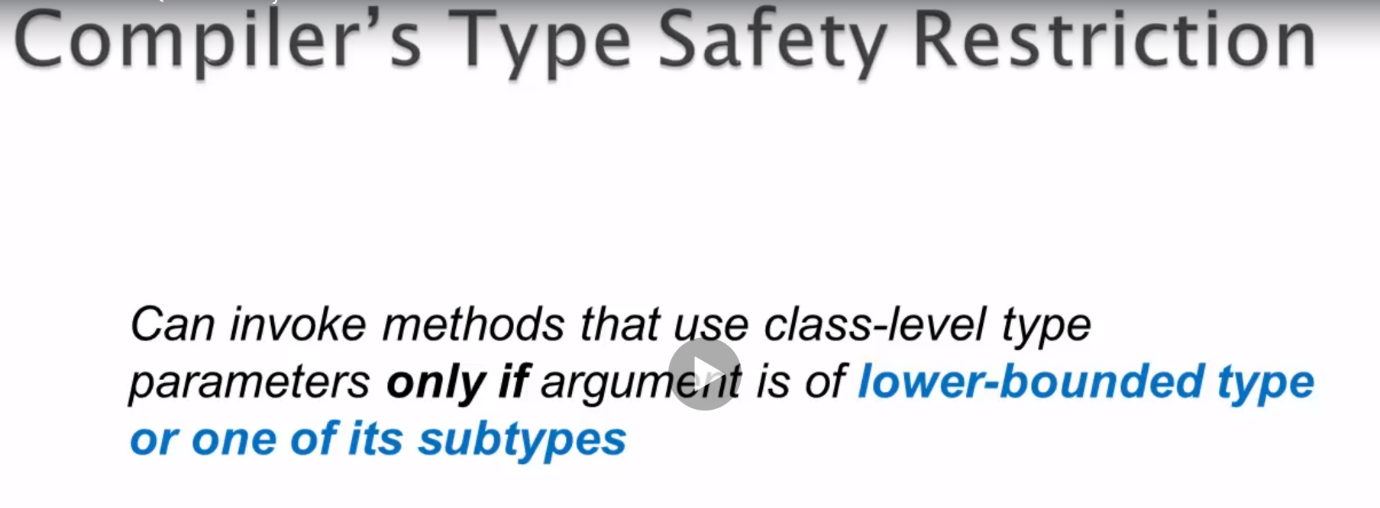


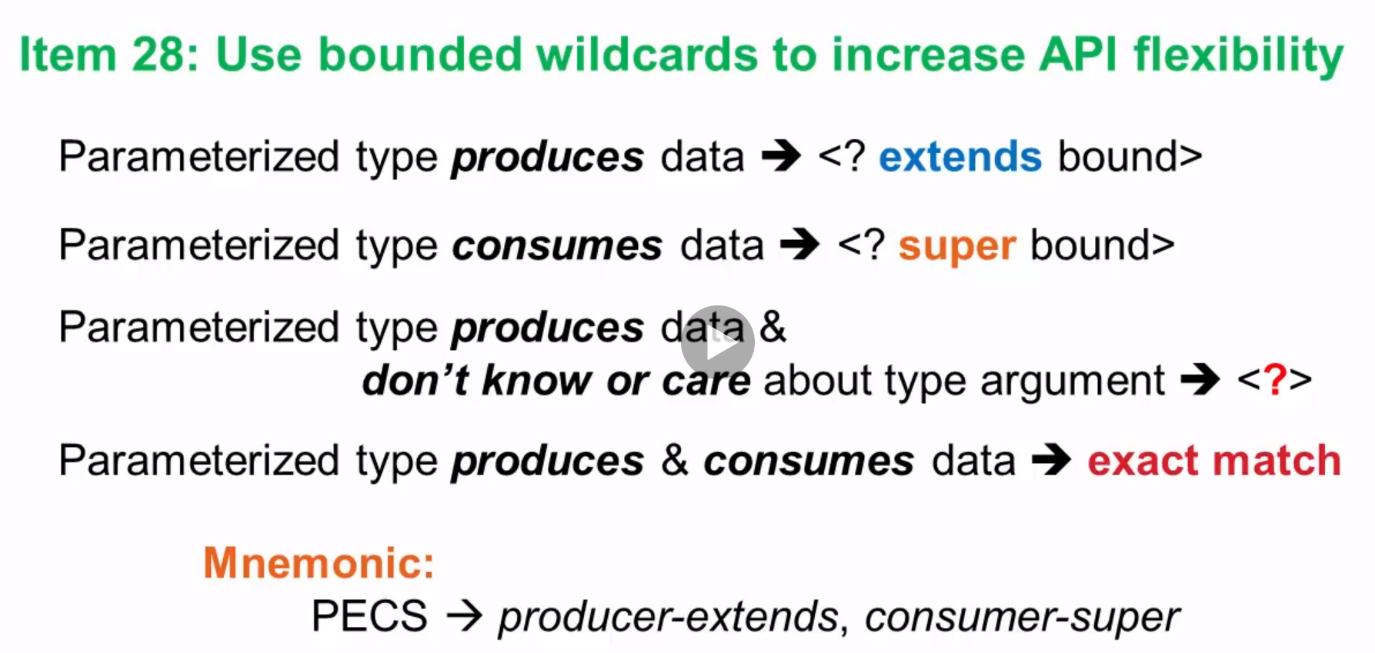




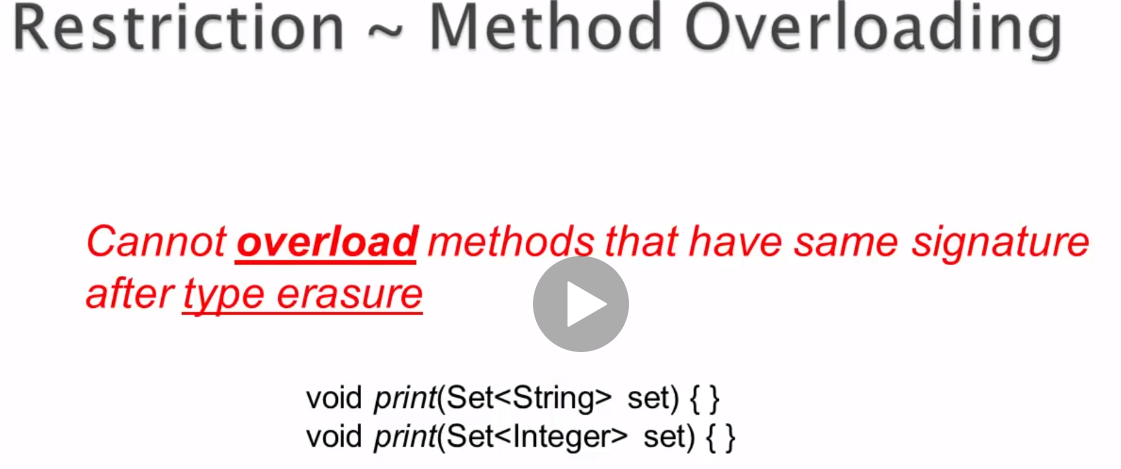




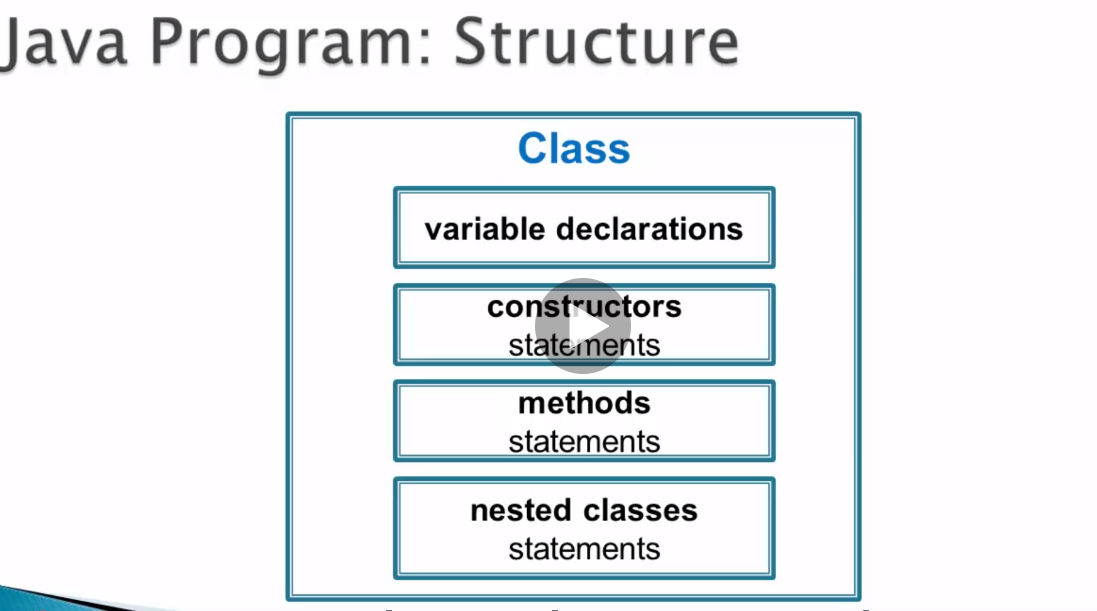




* Class level type parameter can’t be used in static context.
* Type argument can’t be a primitive.
* Generics and arrays don’t mix well.



**Nested classes**



* A class would contain variable declarations, methos, constructors and nested classes.
* Syntax:

class Outer{

class Inner(){

…..}}

Eg:

**Entry Interface [Nested interface]**

**getKey()**

**getValue()**

**Map Interface**

If we invoke entrySet() method on HashMap class, a set of entries will be returned.

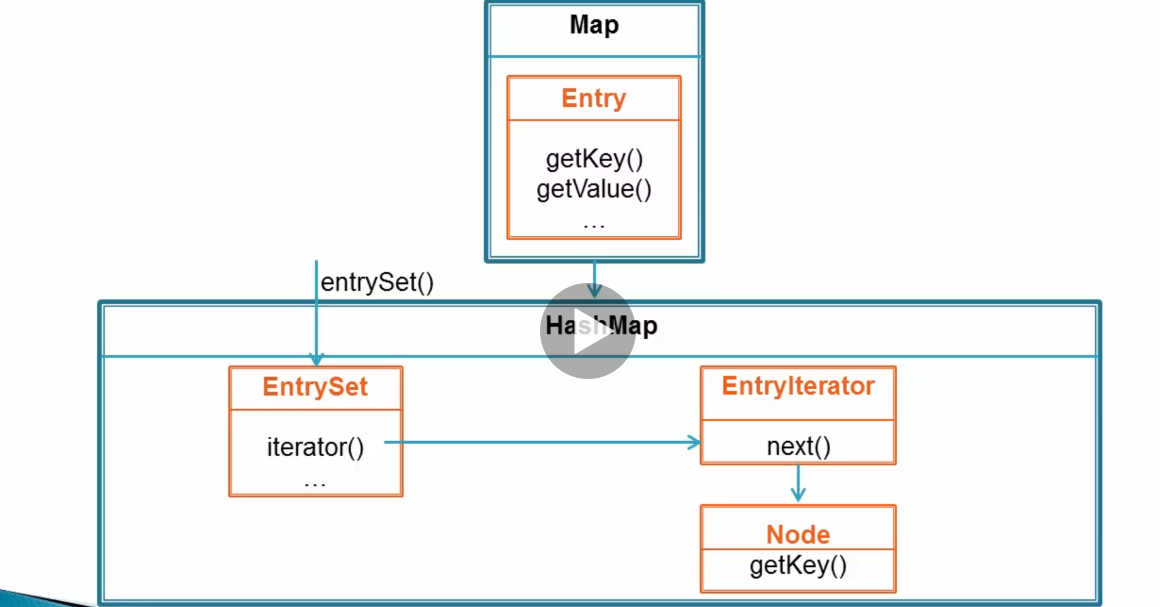
Each entry is implementation of Entry interface.

The returned set is actually nested class within HashMap called “EntrySet” which implements the set interface

If we invoke iterator() method on returned set, it will return an implementation of iterator and that implementation is another nested class within HashMap. It is called “EntryIterator”.

If we invoke “next()” method on EntryIterator,it would give an Entry instance.

This Entry instance is represented by the class called “Node” which is another nested class within HashMap. So, this Node nested class implements Entry nested interface within the Map interface.



**Why can’t we have all these classes as top level classes within the same package as HashMap?**

->There will be 8 additional classes in the package which will only be used by HashMap

This would lead to Class pollution [only HashMap needs it].

* By using nested classes, functionality is kept closer to original class.
* Better design- nested classes modularise the functionality. Without nested classes, the methods contained in them would be a part of enclosing classs. That would lead to method pollution.
* Nested class serves it’s enclosing class

Nested classes types:

1] Inner class – a] Non static member class

B] anonymous class

C] local class

2]Static member class – can never access instance members of enclosing class

Inner class can access instance members. Only exception is if anonymous class or local class is used in static context.

* Nested classes can be declared with all 4 access levels. But, top level class can be declared with only public/ default access levels
* Mutual accessibility of members including private. Outer class can access any member of nested class. Nested class can access any member of outer class.
* Class files will be generated -> Outer.class , Outer$Nested.class.

In the case of anonymous classes, it will Outer$1.class, Outer$2.class

**Non static member class**

* A non static member class is used when inner class object needs access to enclosing class object.
* Enclosing object is created first.
* Inner class object is instance member of enclosed object.
* Inner class object maintains a hidden reference to enclosing class object. Building the hidden reference takes up time and space.
* Enclosing object is not garbage collected until the inner class object is alive.

Common use case:

1] Adapter: Adapters are used to produce different views of outer class object.

Eg: Map-> keyset(), Values(), entrySet()

keyset() returns a “Set” object that contains all keys in the Map. This set object is basically an instance of non static member class which implements “Set” interface. The Set object is basically using the state of the enclosed object. i.e. state would be the keys. Due to this, it requires a hidden reference to the enclosing object.

Eg 2: iterator() found in Set and List interfaces

This method return iterator object which is instance of non static member class which implements “Iterator” interface.

Code:

**public** **interface** CacheIteraor {

**boolean** hasNext();

Bookmark next();

}

**public** **class** Cache {

Bookmark[] items;

**int** count=0;

**public** Cache(**int** size) {

items= **new** Bookmark[size];

}

**public** **void** add(Bookmark item) {

**if**(count< items.length)

{

items[count++]=item;

}

}

**public** CacheIteraor iterator() { //reference type is cacheIterator not MyCacheIterator because MyCacheIterator is private

**return** **new** MyCacheIterator();

}

**private** **class** MyCacheIterator **implements** CacheIteraor{

**private** **int** i=0;

@Override

**public** **boolean** hasNext() {

**return** i< items.length;

}

@Override

**public** Bookmark next() {

**return** items[i++];

}

}

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

Cache recommendedItems =**new** Cache(5);

Bookmark item1=**new** Bookmark();

item1.setId(1);

item1.setTitle("asshu");

Bookmark item2=**new** Bookmark();

item2.setId(2);

item2.setTitle("asshu1");

Bookmark item3=**new** Bookmark();

item3.setId(3);

item3.setTitle("asshu2");

Bookmark item4=**new** Bookmark();

item4.setId(4);

item4.setTitle("asshu3");

Bookmark item5=**new** Bookmark();

item5.setId(1);

item5.setTitle("asshu4");

recommendedItems.add(item1);

recommendedItems.add(item2);

recommendedItems.add(item3);

recommendedItems.add(item4);

recommendedItems.add(item5);

CacheIteraor iterator=recommendedItems.iterator(); // if MyCacheIterator was public and CacheIterator interface was not there, reference type would have been- Cache.MyCacheIterator

// CacheIteraor iterator1=recommendedItems.new MyCacheIterator(); - alternate

**while**(iterator.hasNext()) {

System.***out***.println(iterator.next().getTitle());

}

}

}

**public** **class** Bookmark {

**public** **long** id;

**public** String title;

**public** **long** getId() {

**return** id;

}

**public** **void** setId(**long** id) {

**this**.id = id;

}

**public** String getTitle() {

**return** title;

}

**public** **void** setTitle(String title) {

**this**.title = title;

}

}

Output:

asshu

asshu1

asshu2

asshu3

asshu4

* Under the hood, 2 .class files are created i.e. Cache and Cache$MyClassIterator
* When iterator() is invoked, it will be like this:

**public** CacheIterator iterator(){

**return** **new** Cache$MyCacheIterator(this); //the outer object’s reference is passed. It will use this reference to access outer class members within the inner class

}

* Cannot have static members in non static member class [Static member class object is an instance member of outer class obj. So, static member in non static member class doesn’t make sense].

Access

1] To access inner class method from outer class –

InnerClassobj.Innermethod();

2] To access outer class method from inner class-

A] go() – if go() method is present in inner class, outer class’ go method is invoked.

B] this.go() – go() method of inner class is invoked if present. If not present- compilation error

C] Outer.this.go() – method of outer class

**Anonymous class**

**-> C**lass which doesn’t have a name.

Eg: Let us consider TreeSet constructor which takes Comparator as input. Comparator is an interface which has the abstract method compare which compares two arguments and helps in maintaining the sorting order in the TreeSet.

We can define concrete classes which implements the Comparator interface. However, we can also use anonymous classes to implement Comparator interface.

Eg:

Set set=new TreeSet( new Comparator<String>() { // argument is instance of anonymous class implementing Comparator interface

public int Compare(String s1, String s2){

return s1.length()-s2.length();

}

});

* + We can see that anonymous class is declared and instantiated at the point of use. It is usually used as a method argument.
  + In the above example, the whole expression can be read as “Create an object of anonymous class that is inherited from Comparator”.
  + Compiler will create a separate top level class corresponding to the anonymous class which would implement the Comparator interface. In byte code, there won’t be any anonymous nested class. It will be a top level class. Such anonymous classes would have class names such as “Outer$1” and “Outer$2”. [Generally, OuterClassName$InnerClassName. In case of anonymous class, OuterClassName$Number].
  + Common use case: Function Object.

Function Object:

1. It’s methods operate on other objects. I.e method parameter are other objects. Compare method also operated on 2 input String objects.

2. Function Objects export only one method. Eg: compare()

-> Function object is basically instance of a type with the above 2 properties. The type can be class or interface. If the type is an interface, it is called functional interface.

Comparator interface is considered as functional interface since it has only one abstract method - “compare”. It also has another abstract method - “equals”. Since, sub classes implement the equals method from the object class, it is still considered as functional interface.

* + Item 21: Use function objects to represent strategies. Comparator interface would be referred as Strategy interface. Anonymous classes implementing it would be concrete Strategy classes.
  + **Anonymous classes – Best practices**

**1.** Should be short- 10 lines or fewer.

2. w.k.t anonymous class is declared and instantiated at the point of use. Anonymous class object is created every time the anonymous class expression is being executed. In such cases, we can use private static final field to declare and instantiate the anonymous class.

Eg: private static final Comparator<String> STR\_LEN\_COMPARATOR = new Comparator<String>(){

public int Compare ( String s1, String s2 ){

return s1.length()-s2.length();

}

}; //semi colon is present at the end

-> By this, only one anonymous object is created. It can be reused by using the field name.

In this case, we can pass STR\_LEN\_COMPARATOR as an argument to the TreeSet constructor.

-> The type argument on the right side is emphasized in red. This can’t be auto inferred in the case of anonymous classes.

->One advantage is that we can have a nice descriptive field which tell us what the anonymous class does.

->Anonymous class can have access to the enclosing class if and only if it occurs in a non-static context. I.e. if the anonymous class is declared inside a static method, it can’t access the instance member of enclosing class. The anonymous class object won’t be having access to enclosing class object.

->If anonymous class is declared within instance method, it would be having access to enclosing class object via a hidden reference.

-> Anonymous class can’t have a named constructor. However, it can have instance initializer to initialize variables.

->Anonymous class can also be used to implement a class.

**Code:**

**Bookmark.java**