

Collections Overview

Java collection framework: list, vector, array, linkedlist, sets, maps, trees, queues.

So at the top level of collection framework is the collection class, now it exposes static methods that you can operate on collections such as the sort method.

Interfaces in framework they allow framework to be extended. They define methods for all the fundamental operations that really are required for various collection types.

One of the goal of collection framework there should be good interoperability among various collections, so not just the ones included in the framework.

But literally anything that may also be created in the future, that is a reasonable representation of a collection, now that includes arrays.

Now array themselves couldnt be made part of the collection framework without changing the Java language.

However the framework does include methods that enables collection to be move into arrays and vice versa and additionally methods to allow arrays to be viewed as collections.

Core elements of collections framework are interfaces.

These are the abstract types that represent collection like List interface.

Java jdk provides a reange of polymorphic algorithms that work on collection objects.

```

public class Theatre {
    private final String theatreName;
    private List<Seat> seats=new ArrayList<>();
}

public class Theatre {
    private final String theatreName;
    private List<Seat> seats=new LinkedList<>();
}

We can be really more generic
public class Theatre {
    private final String theatreName;
    private Collection<Seat> seats=new LinkedList<>();
}

```

So the advantage of that is it enables us to use any of the collection classes to hold our seats.

Collection hierarchy : [collection hierarchy](#)



So when we declare seats of type collection, we can implement the list of seats in a theater using any concrete class that implements one of the interface that extend collection.

We can also use Hash set or link hashset that implement sets if we want to.

Reminder the code works cause we did a linear scan.

```
public void getSeats(){  
    for(Seat seat:seats){  
        System.out.println(seat.getSeatNumber());  
    }  
}
```

```
public class Theatre {  
    private final String theatreName;  
    private Collection<Seat> seats=new HashSet<>();  
}
```

Works . Also, the elements in the HashSet are jumbled.

```
public class Theatre {  
    private final String theatreName;  
    private Collection<Seat> seats=new LinkedHashSet<>();  
}
```

Work, Now , the elements are in order.

Although we can use any concrete class that implement set, list,queue,deque.
What we cant do is drop down a level and still expect things to work.

The collection framework also includes a TreeSet, which implements SortedSet

```
public class Theatre {  
    private final String theatreName;  
    private Set<Seat> seats=new TreeSet<>();  
}
```

Error: Exception in thread "main" java.lang.ClassCastException Crea

It has having trouble casting to the comparable interface, which is one level down.

Tree set is below sorted set.

You can use only direct child of Collection framework.

So the TreeSet we tried to use implement SortedSet and actually what it does its navigable set, which extends sorted set. It got an additional requirement that elements in it must contain or must be comparable. Thats How set is Sorted.

Because we didnt make the seat class comparable thats why we got that ClassCastException.

As we move down the hierarchy, the interface becomes more specialized so we can only replace classes with other classes that implement one of the core collection interface at the same level.

{To implement binary search in collection framework we need to implement comparable interface}*
}

```
private class Seat implements Comparable{
```

Here you Comparable to the unit. |

```
int foundSeat= Collections.binarySearch(seats,requestSeat, c: null);|
```

First argument can't be a collection, it has to be atleast a list object
the third parameter is null because we used a in-built compareTo on Seat class.

```
//shallow copy Lec 155 5:00  
// Same address.  
List<Theatre.Seat> seatCopy = new ArrayList<>(theatre.seats);  
printList(seatCopy);
```

```
Collections.reverse(seatCopy);  
Collections.shuffle(seatCopy);  
System.out.println("Printing seatCopy");  
printList(seatCopy); //prints reverse(desc)  
System.out.println("Printing Theatre seats");  
printList(theatre.seats); //prints (ascending)
```

```
Theatre.Seat minSeat = Collections.min(seatCopy);  
Theatre.Seat maxSeat = Collections.max(seatCopy);  
System.out.println("Min seat number is "+ minSeat.getSeatNumber());  
System.out.println("Max seat number is "+ maxSeat.getSeatNumber());|
```

```
public static void sortList2(List<? extends Theatre.Seat> list){  
    for(int times=0;times<=list.size()-2;times++) {  
        for (int i = 0; i <=list.size() -2; i++) {  
            int comparsion = (list.get(i).compareTo(list.get(i + 1)));  
            if (comparsion > 0) {  
                Collections.swap(list, i, i + 1);  
            }  
        }  
    }  
}
```

```
}
```

Collection copy method

It takes two parameters the destination and the source list, the destination has to be of a generic type, there's a super set of the source list I should say and it basically means it can be a collection an iterable or a list and the source list must be a list.

The problem that tends to arise with the collections copy method is that people either expect to use it as we just did when we used the list constructor to create our copy of our seats.

```
Theatre theatre = new Theatre("Olympian", 8, 12);  
List<Theatre.Seat> seatCopy = new ArrayList<>(theatre.seats);
```

Is What happened is people may expect to create a deep copy of list.

As opposed to shallow copy, a deep copy is a copy where the elements are not just references to the same elements as in the original list, but are themselves copied.

```
List<Theatre.Seat> newList = new ArrayList<>(theatre.seats.size());  
Collections.copy(newList, theatre.seats);
```

This will give error, newList just reserve space and its empty.

For Collections copy to work newList must have some seed value.

Comparable and Comparator Interfaces.

Another way of using a sort method, is to pass in a comparator.

Similar to comparable, comparator interface defines a single method called compare.

Unlike comparable, the object to be sorted don't have to implement comparator.

Instead an object of type comparator can be created with a compare method that can sort the objects that we are interested in and more than one comparator can be created and it allows for objects to be sorted in different way which is pretty cool.

So we can either create a comparator object within an existing class or we could create a new class that implements comparator interface.

So let's have a go at the first way using anonymous inner class within Theatre.

```
static final Comparator<Seat> PRICE_ORDER=new Comparator<Seat>() {  
    @Override  
    public int compare(Seat seat1, Seat seat2) {  
        if( seat1.getPrice() < seat2.getPrice()){  
            return -1;  
        }  
        else if( seat1.getPrice()>seat2.getPrice()){  
            return 1;  
        }  
        return 0;  
    }  
};
```

Above Anonymous Inner class Implementing comparator and its providing an implementation of the single compare Method.

Comparator doesn't need to be static, but it makes it easier if we don't need to use class instance in order to use it.

Above compare is implemented in Theatre class and not in Seat class. It could have made more sense to include it in the seat class itself.

Now, instead. Especially if seat was now an inner class.

But had we done that we can't use static method.(without an instance) in Main.

```
List<Theatre.Seat> priceSeats = new ArrayList<>(theatre.getSeats());  
priceSeats.add(theatre.new Seat( seatNumber: "B00", price: 4.00));  
priceSeats.add(theatre.new Seat( seatNumber: "A00", price: 4.0));  
Collections.sort(priceSeats, Theatre.PRICE_ORDER);  
printList(priceSeats);
```

Some issue discussed about comparator, I didnt get it. In lect Maps 157.

Map Interface

Map interface is part of the collection framework even though its not a true collection.

Map interface replaces the obsolete dictionaries abstract class.

Map is like dictionary. It is unordered regardless of sequence of insertion.

Java map cannot contain duplicate keys and each keys can only map to a single value.

Two classes that implement Map interface That is **HashMap** and **LinkedHashMap**.

TreeMap implements the sorted map interface.

Now Map like all the core collection interfaces are Generic, they take two types, one for the key and one for the value. It is possible to use raw maps where the type arent specified.

```
Map<String,String> languages=new HashMap<>();
languages.put("Java"," a compiled level, object-oriented," +
    "platform independent language");
System.out.print(languages.get("Java"));
```

{Maps contains uniquer keys, if u try a key that already exist,the value in (key,value) it will get updated.}*
}

```
6 Map<String,String> languages=new HashMap<>();
7 System.out.println("7"+languages.put("Java"," a compiled
8     "platform independent language"));
16 System.out.println("16"+languages.put("Java","Hey there"));

7 null
16 a compiled level, object-oriented,platform independent language
```

From Above, put method itself can be used to tell if a value is being added for the first time.

It did prevent you from adding, but if you do want to determine an unpragmatically only add a key if its not already there we can do that as well

Note By Me: Basically to enforce immutability or other things


```
if(languages.containsKey("Java")){
    System.out.println("Java is already in the map");
}
else{
    languages.put("Java","this course is about java");
}
```

Method: **putIfAbsent**

```
System.out.print(languages.putIfAbsent("Java","My Name is kazim"));
```

It works as the name says.

If not absent, it returns the same output as

```
languages.get("Java")
```

Thats really intended to prevent concurrency issues that one thread does not add to the the map only for that entry to be overwritten by another thread. It doesnt help in Null case bcoz it will overwrite the keys with no values.

--To loop through all items in Map use **keySet** method, it returns a set of all keys.

Method: **keySet**

```
for(String key: languages.keySet() ){
    System.out.println(key+" : "+languages.get(key));
}
```

Method: **remove**

```
languages.remove( key: "Lisp");
//remove a key only if its map to a certain value
if(languages.remove("Java","Hey there")){
    System.out.println("Java removed");
}
else{
    System.out.println("Java not removed");
}
```

It returns True or False.

Method: **replace**

Replace the entry for a specified key if its already mapped to a value.

```
languages.replace("Python", "a good language");  
languages.replace("kaju", "a Sweet food");
```

Output:

```
Basic : Beginners All purposes Symbolic Instructioncode  
Python : a good language  
Algol : an algorithmic language
```

Just like `remove()`, we can also specify existing value so the mappings only updated if the key is map to the Old value.

```
if(languages.replace( key: "Java", oldValue: "Hey there", newValue: "Did it replaced?")){  
    System.out.println("Java Replaced");  
}
```

Use case of above, for e.g if you are updating someone's name after they get married by doing so you are making sure the correct person is updated.

The methods defined by the map interface we've used, the map uses a string type for both the key and value.

Infact both the key and values can be any object, we could even use a map type as the values in another map. As an actual fact you can add a map as a value to itself.

Unlike other languages such as Python for e.g there's no requirement that the keys in map be immutable.

{When we assign new value to a String variable, what we are doing we're changing the value the value that the variable holds, we are not changing the String}*
}

Algol for e.g cant be changed, on the other hand array list can change by having new items added when a new value is assigned to existing entries.

Java doc warns that great care must be exercised if immutable objects are used as map keys so they give you that warning.

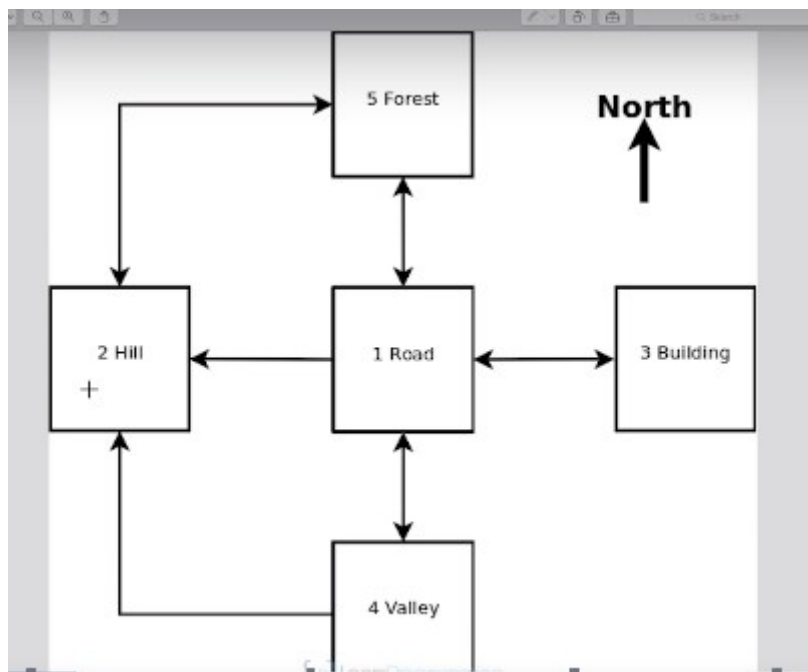
The reason for the warning is map object can behave unpredictably if the key objects change in such a way that equals comparisons are affected.

Trick

```
public class Location {  
    private final int locationId;  
    private final String description;  
    private final Map<String, Integer> exits;  
  
    public Map<String,Integer> getExits(){  
        return new HashMap<String,Integer>(exits);  
    }  
}
```

Above instead of returning of **exits** map, I am returning a new Hashmap with **exits** data.

This is useful cuz nothing outside of the class can change **exits**;



Map literals are not available in java, as a result we have to use repeated calls to add exit for each location.

```
//shallow copy Lec 155 5:00
// Same address.
List<Theatre.Seat> seatCopy = new ArrayList<>(theatre.seats);
printList(seatCopy);

Collections.reverse(seatCopy);
System.out.println("Printing seatCopy");
printList(seatCopy); //prints reverse(desc)
System.out.println("Printing Theatre seats");
printList(theatre.seats); //prints (ascending)
```

Above, Tim reserved a seat using a shallow copy

Then

when we used a Theatre object to reserve seat it gave us prompt " Seat is already reserved"

Reference: Just a piece of address on a Note.

What is Shallow copy: Above shown

What is deep copy?

As opposed to shallow copy , a deep copy is a copy where the elements are not just references to the same elements as in the original list, but are themselves copied.

-----Tim words

{When we assign new value to a String variable, what we are doing we're changing the value that the variable holds, we are not changing the String}

----- Tim words

The rest is my intuition what is going on here?

$[0, \Omega, \square, \star]$
→ $[\text{ref}(0), \text{ref}(\Omega), \text{ref}(\square), \text{ref}(\star)]$

This is mutable.

You can Add, ~~or~~ ~~Remove~~
any element

Considering it's a shallow copy
for oth item object if you change
the state of its field.

for e.g. Circle.PI
It will affect them both.

But in the eg, we are storing
string & Integer

[$\text{ref}(0)$, $\text{ref}(1)$, $\text{ref}(0)$, $\text{ref}(1)$]

so if you update this

0th element \rightarrow ~~$\text{ref}(0)$~~
 \rightarrow "Sup"

Here, the pointer is just swap.

```
locations.get(1).addExit( direction: "N", location: 5);  
locations.get(1).addExit( direction: "Q", location: 0);
```

```
locations.get(2).addExit( direction: "N", location: 5);  
locations.get(2).addExit( direction: "Q", location: 0);
```

```
locations.get(3).addExit( direction: "W", location: 1);  
locations.get(3).addExit( direction: "Q", location: 0);
```

```
locations.get(4).addExit( direction: "N", location: 1);  
locations.get(4).addExit( direction: "W", location: 2);  
locations.get(4).addExit( direction: "Q", location: 0);
```

```
locations.get(5).addExit( direction: "S", location: 1);  
locations.get(5).addExit( direction: "W", location: 2);  
locations.get(5).addExit( direction: "Q", location: 0);
```

```
public Location(int locationId, String description) {  
    this.locationId = locationId;  
    this.description = description;  
    this.exits=new HashMap<String,Integer>();  
    this.exits.put("Q",0);  
}
```


Immutable Classes

Immutable classes means they can't be changed once they are created.

The technique is valuable if you want instances of your class to be immutable but also using some of the techniques in immutable classes is a great way to increase encapsulation and reduce errors even if you're going to allow external code to modify your class instances.

Imagine modern software that supports third-party extensions or plug-ins now if your IntelliJ idea for e.g. support other languages such as Python by providing the plugins that can be developed by third party.

Many modern browsers is another e.g. that support for third-party plugin for years.

And even games allow external developers to create additional rooms or levels so another example is Microsoft Office which provides access to its objects to macros written in VBA

Now all of these examples, exposed programs' inner objects to external developer who don't have access to the source code and therefore they can't know the implication of changes they might make to objects if they are allowed to do that.

So reading the API documentation will help them but it's reasonable for them to assume that if they're permitted to change fields and properties then they can go ahead and do so.

So rather than returning the exits object directly which would expose it to the potential for being changed externally we created a new HashMap object that contains all the elements of exits and return that.

So external code that needs to use the exits when we displayed the list of available exits to the player in the main class to do so there's no chance of changing our internal map, also we could in fact return the list of available exits as a string which maybe more convenient for our particular program but that would have reduced flexibility in it that kind of approach is taken as rule, then we have to consider all the possible uses that external code may actually want to make from our map.

So approach `getExit` doesn't restrict external code from doing what it needs to do with a map it just prevents that from changing our map which is obviously good from our point of view.

2nd technique, By using **Final** keyword for Map, the field shouldn't change and it's not like we forgot to write a setter

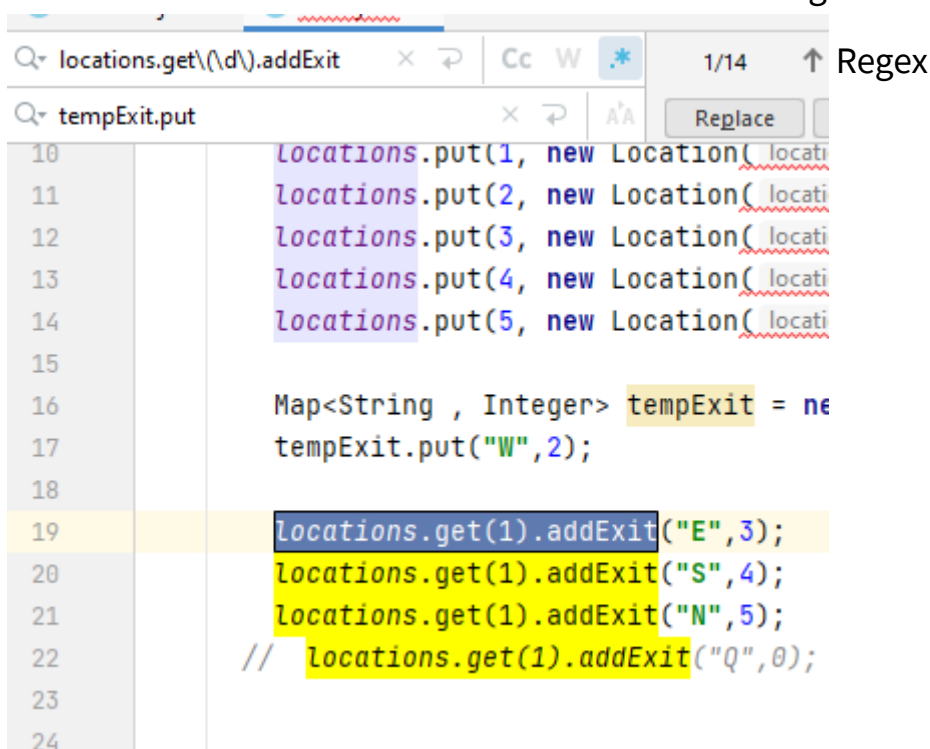
Also ensure that we don't inadvertently change the fields if we change the code in the location class at some time in future.

Tip: Don't provide setter along with getter, only if there's a need.

Sometimes it makes sense to leave the setter in our class so that if someone else wanted to extend the game they could add new exits from existing locations to the new ones.

So the technique for setting the initial exits then preventing further change so if a class field or elements of an immutable class like a list or a map is to be set only once and never changed then what it needs to be done and where it needs to be set is in the constructor.

So we gonna modify the class constructor to accept the map object and will change our constructor signature and assign the passed map to our class field and that means we can delete the `addExits` Method all Together.



```
10 locations.put(1, new Location( locati
11 locations.put(2, new Location( locati
12 locations.put(3, new Location( locati
13 locations.put(4, new Location( locati
14 locations.put(5, new Location( locati
15
16 Map<String , Integer> tempExit = ne
17 tempExit.put("W",2);
18
19 locations.get(1).addExit("E",3);
20 locations.get(1).addExit("S",4);
21 locations.get(1).addExit("N",5);
22 // locations.get(1).addExit("Q",0);
23
24
```

<https://www.udemy.com/course/java-the-complete-java-developer-course/learn/lecture/4152148#questions/8773706>

For immutable class: [link](#)

- No field will change after constructor is done.
- Also try to encapsulate its field.
- dont put setters along with getters.(in rush, only if theres a need)
- In the lec, we used a Map object in our Constructor, so create a shallow copy of it, so even though in Main some entry of Map object got deleted, we still have one reference.

Now,

Depending on the functionality class must provide it may not be possible to employ all these techniques they should certainly be considered and you should be doing that when you're creating classess considering these things in as many of them as possible used in order to increase encapsulation and reduce error.

Link: [immutable object](#)

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields `final` and `private`.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as `final`. A more sophisticated approach is to make the constructor `private` and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
 - Don't provide methods that modify the mutable objects.
 - Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

Its not just overriding methods that can prevent our class from being immutable, adding new methods that expose our map will also do.

So dont allow ur class to be subclassed.

But if you Location type for the key instead of the value, we wanted to ensure that only completely immutable objects could be used as the keys, then we have to prevent subclasses over which we have got no control from being used.

Composition is possible but not inheritance. So using composition they create methods that then call the location class methods from an instance of a location.

They would be able to almost everything that's subclassing would have allowed them to do except use their own class where location object was expected and most of the time, you do not want ur classes to be immutable.

Even if u dont want immutable class, you can use this techniques in various combinations to make your class certainly more robust.

Semantic error: which lead to a program to compile but crash in runtime.

```
public Location(int locationId, String description, Map<String,Integer> exits) {  
    this.locationId = locationId;  
    this.description = description;  
    this.exits=new HashMap<String,Integer>(exits);  
    this.exits.put("Q",0);  
}
```

Above,

The problem here is the Constructor will crash with a NullPointerException.

If we passed null in place of Map exits;

```
public Location(int locationId, String description, Map<String,Integer> exits) {  
    this.locationId = locationId;  
    this.description = description;  
    if (exits != null) {  
        this.exits=new HashMap<String,Integer>(exits);  
    }  
    else{  
        this.exits.put("Q",0);  
    }  
}
```

Sets & HashSet

List: can contain duplicate and have an order

set: Only unique items and doesn't have an order.

Set interface defines the basic method:

add, remove, clear, size, isEmpty, contains, keyset, addAll(bulk insertion)

Interesting enough, there is no way of retrieving item from a set, you can check if something exists and iterate over all the elements in the set.

Using immutable object as elements in a set can cause problems and the behavior is undefined if changing an object affects equal comparisons.

The best performing implementation of set interface is HashSet Class and that uses and that uses hashes to store the items.

It's just like HashMap class.

Like a set can be implemented using a map and it's not hard to work out that.

A good case can be built where a field for identifying the body as planet, moon, comet etc. The set will build by iterating through the entire solar system and checking the destination before adding each body to the appropriate set. Adding moon to a body for e.g. Jupiter would be quite complicated.

Tip:

we used a Map<String, HeavenlyBody>

HeavenlyBody store kb info on each body but faulty reasoning

is it's more efficient to just store the String name in set rather than whatever collection type you using, but a reference to a string is exactly the same size as a reference to any other object so nothing gained and the code becomes more complex because the actual object has to be retrieved from the map b4 you can get any of its details.

So that's an important consideration

now if the id of the object was an integer rather than a string then it might be tempting to store that in the set but this is also false optimization.

So on a 32 bit Java Virtual Machine an int or an object reference both take 4 bytes. I.e 32 bits.

On 64 bit machine the object reference would only be 64 bits, but java use compressed pointer that end up using less space than the 32 bit int.

Link: [Compressed OOP pointers](#)

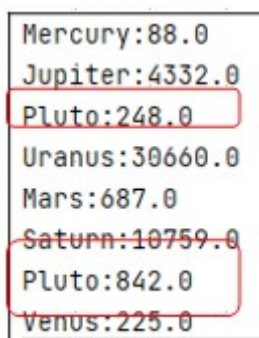
Lec-164:

HashSet – equals() and hashCode()

Java documentation does warn you if you're using your own objects as either a key in a map or as an element in a set that you should override the equals and hashCode methods.

```
public class Main {  
    private static Map<String, HeavenlyBody> solarSystem = new HashMap<>();  
    private static Set<HeavenlyBody> planets = new HashSet<>();  
    public static void main(String[] args){
```

```
        HeavenlyBody pluto = new HeavenlyBody( name: "Pluto", orbitalPeriod: 842);  
        planets.add(pluto);
```



Mercury:88.0
Jupiter:4332.0
Pluto:248.0
Uranus:30660.0
Mars:687.0
Saturn:10759.0
Pluto:842.0
Venus:225.0

Pluto is shown in a Set

Depending on ur OS and version of java, the order is shown.

The reason is the two java object do not compare equal so the set is happy to accept both of them. So @Override equals methods.

The reason that they dont compare equal is because the base object class from which all other classes are derived just defines a very simple equals method that

performs what is known as referential equality and both references point to the same object then their equal otherwise they are not.

Note: it basically does == which does referential equality

so thats why we do this "kazim".equals("kazim");

```
Object o=new Object();
o.equals(o);
"pluto".equals("");

public boolean equals(Object obj) {
    return (this == obj);
}
```

Basic referential equality

which basically says both references point to the same object they are equal.

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
```

```

1021  @ public boolean equals(Object anObject) {
1022      if (this == anObject) {    Checking If string is compared to
1023          return true;          itself
1024      }
1025      if (anObject instanceof String) {    Is a String
1026          String aString = (String)anObject;
1027          if (!COMPACT_STRINGS || this.coder == aString.coder)
1028              return StringLatin1.equals(value, aString.value);
1029          }
1030      }
1031      return false;
1032  }

```

The String Class implements object method and over-rides the equal method

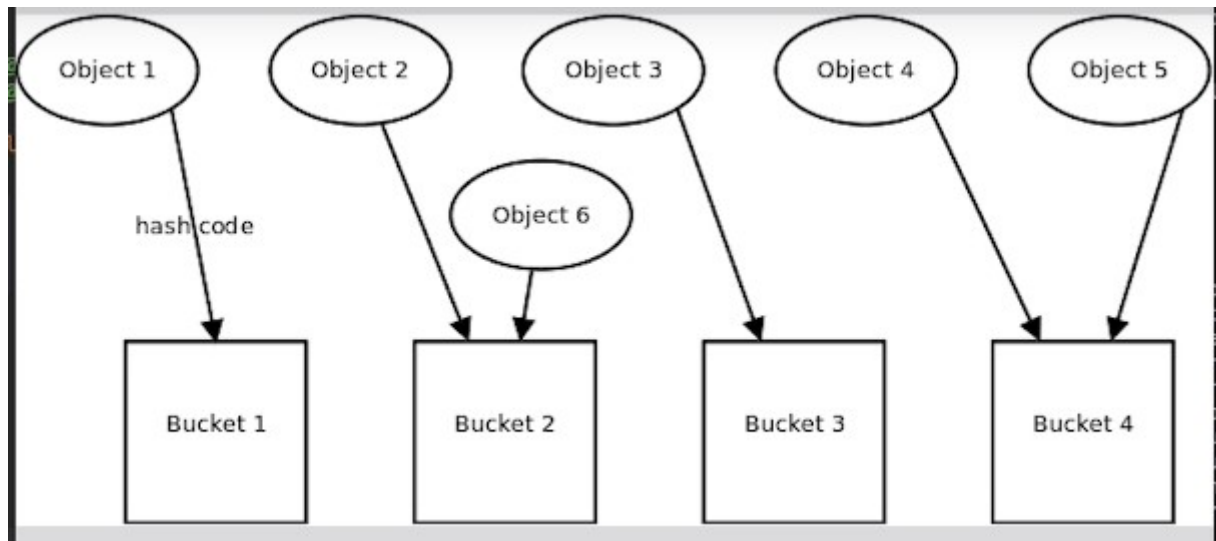
Above at line 1025, if our Heavilybody is subclassed , instance of it will return true when an object of subclass type is used here. We need to look at alternative check to perform when we implement our Heavilybody class.

```

94  @ public static boolean equals(byte[] value, byte[] other) {
95      if (value.length == other.length) {
96          for (int i = 0; i < value.length; i++) {
97              if (value[i] != other[i]) {
98                  return false;
99              }
100          }
101          return true;
102      }
103      return false;
104  }

```

When two object compares equal then their hash codes must also be the same, so u must @Override the **hashCode()** method when u @Override **equals**.



When storing Objects in a hash collection such as `HashSet` or `HashMap`, think of collection as number of buckets to store the objects in.

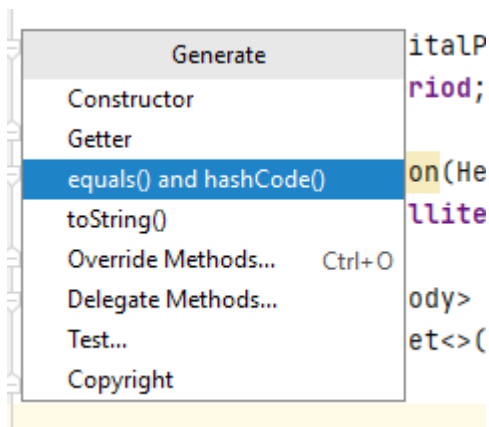
The `hashCode()` (the hash) determine which bucket the object will go into. There's a requirement any object that are equal should always have the same hash code and Ultimately end up in the same bucket. Opposite are not required. Two object which are equal do not have to have different hash code.

There maybe already elements in the bucket, so each is compared to the new object to make sure its not already in there. Because the comparisions is performed using the `equals` method, the collection will know if its looking in the right bucket.

So the `hashCode` must be the same and `equals` return true.

Its no use if `equals` will return true if its checking the wrong bucket.

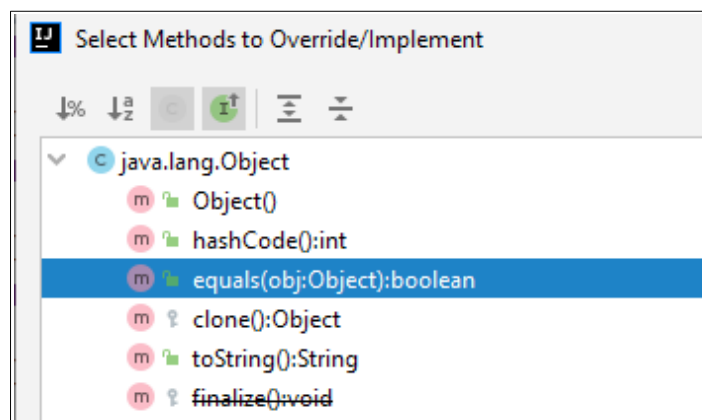
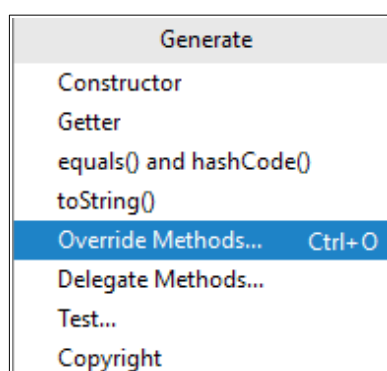
If we were to add new Object that is equal to object 6, the `hashCode` will indicate butcket 2 is to check. It is compared(linear scan) and if its equals the new object isnt added else otherwise.



Or you could do manually, but pls add annotation. Make sure you are Overriding and not overloading.

```
@Override
public boolean equals(HeavenlyBody obj){
    if(this==obj){
        return true;
    }
    System.out.print("obj.getClass() is"+ obj.getClass());
    System.out.print("this.getClass() is"+ this.getClass());
    if( (obj==null) || (obj.getClass()!=this.getClass())){
```

To Overcome @Override annotation error.



Link: [Rules for equal](#)

Summary:

Reflexive, symmetric, transitive, consistent

So now u know what parameters should be to @Override and make changes accordingly.

```
@Override
public boolean equals(Object obj){

    if(this==obj){
        return true;
    }
    System.out.println("obj.getClass() is"+ obj.getClass());
    System.out.println("this.getClass() is"+ this.getClass());
    if( (obj==null) || (obj.getClass()!=this.getClass())){
        return false;
    }
    String objName=((HeavenlyBody) obj).getName();
    return this.name.equals(objName);
}
```

And now the error is gone.

```
@Override
public int hashCode() {
    return super.hashCode();
}
```

String hashcode method

```
@Override
public int hashCode() {
    return this.name.hashCode();
}
```

Heavilybody hashcode method.

Tip: you have to fix both hashcode and equals method for just equals to work , Cuz even if equals is fixed, pluto is placed in a different bucket, so we have duplication.

```
@Override
public int hashCode() {
    return this.name.hashCode()+57;
}
```

we adding 57 to get a non zero number.

```
Uranus:30660.0
Mercury:88.0
Pluto:248.0
Earth:365.0
Jupiter:4332.0
Mars:687.0
Neptune:165.0
Saturn:10759.0
Venus:225.0
```

The problem is solved.

Video: Potential issue with equals() and subclassing

```
public class Challenge {  
    /**  
     * When overriding the equals() method in the HeavenlyBody class, we  
     * were careful to make sure that it would not return true if a HeavenlyBody  
     * was compared to a subclass of itself.  
     *  
     * We did that to demonstrate that method, but it was actually  
     * unnecessary in the HeavenlyBody class.  
     *  
     * The mini challenge is just a question: why was it unnecessary?  
     *  
     * Hint: If you are stuck, check out Lecture 97  
     *  
     */  
}
```

Lec 97: Hamburger challenge

Only tip: if u have base class which has some public methods, and u create a subclass from the baseclass. And u dont want those public methods for your subclass, so you just @Override those methods and delete the code in it and if u like, u can prompt “method not accessible” or in our e.g cannot add additional items to a deluxe burger.

```
* The HeavenlyBody class is declared final, so cannot be subclassed.  
* The Java String class is also final, which is why it can safely  
* use the instanceof method without having to worry about  
* comparisons with a subclass.
```

I already know that.

Final means u cant subclassed, and private constructor means u cant instantiate. But you can create a subclass of a class who cant instantiate and instantiate that. But you dont want others to do that so u made the class final.

An e.g where u can subclassed and its equal method Overriden.

```

public class Dog {
    private final String name;

    public Dog(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public boolean equals(Object obj) {
        if(this == obj) {
            return true;
        }

        if(obj instanceof Dog) {
            String objName = ((Dog) obj).getName();
            return this.name.equals(objName);
        }
    }
}

```

```

public class Labrador extends Dog {
    public Labrador(String name) {
        super(name);
    }

    @Override
    public boolean equals(Object obj) {
        if(this == obj) {
            return true;
        }

        if(obj instanceof Labrador) {
            String objName = ((Labrador) obj).getName();
            return this.getName().equals(objName);
        }

        return false;
    }
}

```

```

public class DogMain {
    public static void main(String[] args) {
        Labrador rover = new Labrador("Rover");
        Dog rover2 = new Dog("Rover");

        System.out.println(rover2.equals(rover));
        System.out.println(rover.equals(rover2));
    }
}

```

```

true
false

```

```

Process finish

```

true: Labrador is an instance of Dog.

False: Dog isn't an instance of Labrador.

Now if we,

```
public class Labrador extends Dog {  
    public Labrador(String name) {  
        super(name);  
    }  
  
    // @Override  
    public boolean equals(Object obj) {  
        if(this == obj) {  
            return true;  
        }  
  
        if(obj instanceof Labrador) {  
            String objName = ((Labrador) obj).getName();  
            return this.getName().equals(objName);  
        }  
  
        return false;  
    }  
}
```

```
/Library/Jav  
true  
true
```

So, we just make the method final so it cant be Override.

```
public class Dog {  
    private final String name;  
  
    public Dog(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    @Override  
    public final boolean equals(Object obj) {  
        if(this == obj) {  
            return true;  
        }  
  
        if(obj instanceof Dog) {  
            String objName = ((Dog) obj).getName();  
            return this.getName().equals(objName);  
        }  
  
        return false;  
    }  
}
```

If u allow ur classes to be subclassed if so how u treat comparsion between a subclass and its baseclass.

So if the subclass would reasonably only add methods that do not alter the way equal should work then allow subclassing but make ur **equal** and **hashCode()** methods as final.

If a subclasses is likely to be different objects then used the method we use in heavenly body to make comparisions between its subclass and its base class (obj.class == this.obj).

If neither of this really fit then prevents subclassing and force clients to use composition instead.

Sets – Symmetric & Asymmetric

Link: [Set interface](#)

Summary

Set Interface Bulk Operations

Bulk operations are particularly well suited to Sets; when applied, they perform standard set-algebraic operations. Suppose *s1* and *s2* are sets. Here's what bulk operations do:

- *s1.containsAll(s2)* — returns true if *s2* is a **subset** of *s1*. (*s2* is a subset of *s1* if set *s1* contains all of the elements in *s2*.)
- *s1.addAll(s2)* — transforms *s1* into the **union** of *s1* and *s2*. (The union of two sets is the set containing all of the elements contained in either set.)
- *s1.retainAll(s2)* — transforms *s1* into the intersection of *s1* and *s2*. (The intersection of two sets is the set containing only the elements common to both sets.)
- *s1.removeAll(s2)* — transforms *s1* into the (asymmetric) set difference of *s1* and *s2*. (For example, the set difference of *s1* minus *s2* is the set containing all of the elements found in *s1* but not in *s2*.)

To calculate the union, intersection, or set difference of two sets nondestructively (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms.

```
Set<Type> union = new HashSet<Type>(s1);  
union.addAll(s2);
```

```
Set<Type> intersection = new HashSet<Type>(s1);  
intersection.retainAll(s2);
```

```
Set<Type> difference = new HashSet<Type>(s1);  
difference.removeAll(s2);
```

Interaction between Collection framework and arrays.

Collection interface suggest that all Classes that implement collection should provide two standard constructors the first one is an empty constructor, the other one takes collection argument to initialize new collection with all the items in the collection that is passed to the constructor.

1.

```
Set<Integer> cubes = new HashSet<>();
```

2.

```
Set<Integer> union = new HashSet<>(squares);
```

This is a suggestion since interface do not have constructor and this behavior cant be enforced for that reason but all the current collection implementations do this currently and we've used it.

-So the Array class also provides an alist method that is used to return a list view of the elements in the array. In fact it uses an ArrayList.

```
words.addAll(arrayWords);  
for(String s: words)  
    System.out
```

Required type: Collection <? extends java.lang.String>

```
Set<String> words = new HashSet<>();  
String sentence = "one day in the year of the fox";  
String[] arrayWords = sentence.split(regex: " ");  
words.addAll(Arrays.asList(arrayWords));  
for(String s: words){  
    System.out.println(s);  
}
```

words.addAll() takes a Collection object, so we create an List object using Arrays.asList() method while takes arrays.

As you can see in Arrays class

```
public static <T> List<T> asList( @NotNull T... a) {  
    return new ArrayList<>(a);  
}
```

This method related to Array, this method is considered as a bridge between two Api. The asList method provides a convenient way to initialize the collection with a list of values as there is not set of literals in java.

What is symmetric difference?

Imagine two circle intersection, symmetric difference is the elements is found in one of the circle but not in both.

Symmetric difference is

A and B is $\rightarrow |A| + |B| - |A \text{ intersection } B|$

Asymmetric difference is

A and B is $\rightarrow |A| - |B|$ Maybe bcoz of no Reflexivity.

So in sets code:

1. first take union I.e addAll()
2. take intersection using retainAll()
3. union.removeAll(intersection)

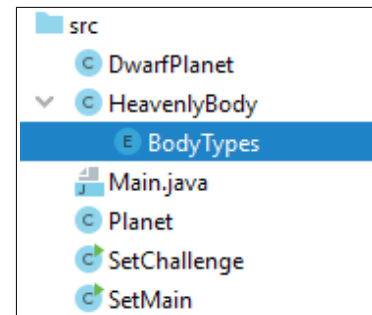
Tip: containsAll() method is non-destructive. It returns True or False

Set Challenge

```
public final class HeavenlyBody {  
    private final String name;  
    private final double orbitalPeriod;  
    private final Set<HeavenlyBody> satellites;  
    private final int bodyType;  
  
    public static final int STAR = 1;  
    public static final int PLANET = 2;  
    public static final int DWARF_PLANET = 3;  
    public static final int MOON = 4;  
    public static final int COMET = 5;  
    public static final int ASTEROID = 6;  
}
```

Java has got a way of grouping constant together, in what's called an enum, so instead of having individual declaration for each.

```
public final class HeavenlyBody {  
    private final String name;  
    private final double orbitalPeriod;  
    private final Set<HeavenlyBody> satellites;  
    private final int bodyType;  
  
    public enum BodyTypes {  
        STAR,  
        PLANET,  
        DWARF_PLANET,  
        MOON,  
        COMET,  
        ASTEROID  
    }  
}
```



```
HeavenlyBody pluto = new HeavenlyBody( name: "Pluto", orbitalPeriod: 842, HeavenlyBody.BodyTypes.PLANET);
```

Advantage of using an enum, is enums are types and it means that when we change the constructor we can make sure that it only accepts valid body type.

```

public final class HeavenlyBody {
    private final String name;
    private final double orbitalPeriod;
    private final Set<HeavenlyBody> satellites;
    private final BodyTypes bodyType;

    public enum BodyTypes {
        STAR,
        PLANET,
        DWARF_PLANET,
        MOON,
        COMET,
        ASTEROID
    }

    public HeavenlyBody(String name, double orbitalPeriod,
                        BodyTypes bodyType) {
        this.name = name;
        this.orbitalPeriod = orbitalPeriod;
        this.satellites = new HashSet<>();
        this.bodyType = bodyType;
    }
}

```

firstly, without our integer constant that we added before changing the body to to an enum, there won't be any compile time checking that the integer body type was one of the values that the integer body type that the class knows about.

Now we have to include code in the constructor to check that, the best we could do is run time error but the advantage of using an enum is we get an error at compile time if body isn't valid so if someone tries to sort of pass an invalid type (star, plane, ...)

We are going to get an error, that's one great Advantage of using an enum.

- A nested enum is automatically static so it's possible to refer to enum values without creating an instance of a heavenly body object.

```

private final BodyTypes bodyType;

public enum BodyTypes {
    STAR,
    PLANET,
    DWARF_PLANET,
    MOON,
    COMET,
    ASTEROID
}

```

This make it a constant

This make it static

```
public boolean addMoon(HeavenlyBody moon){  
    return this.satellites.add(moon);  
}
```

--> By having enum, you can put a type check

```
public boolean addMoon(HeavenlyBody moon){  
    if(moon.getBodyType()==BodyTypes.MOON){  
        return this.satellites.add(moon);  
    }  
    else{  
        return false;  
    }  
}
```

//although the tim program, it was consider to left as it is, just bcoz of Some theory. This approach didnt fit how solar system is shaped. Thats why.

Cuz we restricting that only moons orbit, so he even rename the method to **addSatellite()**.

```

47 public final boolean equals(Object obj){
48
49     if(this==obj){
50         return true;
51     }
52     // System.out.println("obj.getClass() is"+ obj.getClass());
53     // System.out.println("this.getClass() is"+ this.getClass());
54     // if( (obj==null) || (obj.getClass()!=this.getClass())){
55     //     return false;
56     // }
57     if(obj instanceof HeavenlyBody){
58         HeavenlyBody theObject = (HeavenlyBody) obj;
59         if(this.name.equals(theObject.getName())){
60             return this.bodyType== theObject.getBodyType();
61         }
62     }
63     // String objName=((HeavenlyBody) obj).getName();
64     // return this.name.equals(objName);
65     return false;
66 }
67

```

Line 57: checking its an instance of HeavenlyBody.

Line 59: checking the names are same.

Line 60: checking we are comparing planet to planet, stars to stars, moon to moon.

```

@Override
public int hashCode() {
    return this.name.hashCode()+57;
}

```

if we had used integers for our constants, we could have just use the integer values just added that to the 57 but fortunately its not actually much harder with enums cuz enums constants has a hashCode method as well.

```

@Override
public int hashCode() {
    return this.name.hashCode()+57+ this.bodyType.hashCode();
}

```

Note:-Now, planet kazim and asteroid kazim can be placed in different buckets, while maintaing hash characteristics.

The final method is optional but it makes the printing heavilybody easier. We are going to override the **toString()** method so it displays the body type and orbital period as well as the name.

```
public String toString(){
    return this.name+" : "+ this.bodyType+", "+ this.orbitalPeriod;
}
```

```
for(HeavenlyBody planet : planets){
    System.out.println(planet);
    //System.out.println(planet.getName()+" : "+ planet.getOrbi);
}
```

```
Pluto : PLANET, 248.0
Jupiter : PLANET, 4332.0
Saturn : PLANET, 10759.0
Uranus : PLANET, 30660.0
Mars : PLANET, 687.0
```

So Advantage of enum:

-If would have use a constant int then we have to replace with the String

for e.g -----1 with Planet

Note: when we use an object and string concatenation like above, java implicitly called the toString() method to obtain the String representation of the object.

Recap Note: the reason we dont have duplicate pluto is bcoz of set functionality, which help us. But for that to work .we have to make sure the comparsion is based on name field('pluto','mars','jupiter'....) and not referential equality.

.....

Its better to make heavenly body abstract.

Since pluto can be add as a Dwarf planet and Planet.

--primary key

--ssn(using a field)

--name+bodytype

--tim sol

is create a class a key class and give the heavenly body class a field of type key. Because this key is gonna be closely tied to the heavenly body class I am gonna make it a static inner class of the heavenly body class.

```
} public String toString(){
|     return this.name+" : "+ this.bodyType+", "+ this.orbitalPeriod;
| }
| //static class is like a independedent
| // normal class nested, which you
| // can create instance without any association to outerClass
| //contrast to non-static nested class aka Inner class
| // static here means u cant have access to Enclosing class
| // instance variables.
| // can have access to static variables
| // have its own instance variables.
| //Also which can either be private,public,
| // package-private, protected.
| public static final class key{
|     private String name;
|     private BodyTypes bodyType;
| }
}
```

```

public static final class Key{
    private String name;
    private BodyTypes bodyType;

    public Key(String name, BodyTypes bodyTypes){
        this.name=name;
        this.bodyType=bodyTypes;
    }
    public String getName(){
        return name;
    }
    public BodyTypes getBodyType(){
        return bodyType;
    }
    @Override
    public int hashCode() {
        return super.hashCode()+57+this.bodyType.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        Key key=(Key) obj;
        if(this.name.equals(key.getName())){
            return (this.bodyType == key.getBodyType());
        }
        return false;
    }
}

```

Bcoz we gonna be using key in a map thats why we need to Over-ride **equal** and **hashcode** method.

These methods are pretty identical to the **equal** and **hashcode** methods in HeavenlyBody.

Generally speaking its a bad idea to have duplicate code like this.

Note:

All Moons

HeavenlyBody\$Key@2d5686a6
HeavenlyBody\$Key@342f77cc
HeavenlyBody\$Key@22720731

@Override

```
public String toString() {  
    return this.name + " : " + this.bodyType;  
}
```

benefits to Over-ride **toString()**

All Moons

Ganymede : MOON
Phobos : MOON
Europa : MOON

Sorted Collections

variant of HashMap and HashSet classes: **Linked Hashmap** and **Linked hashSet** .

There are also sorted version of Map and set interfaces that are implemented in **TreeMap** and **TreeSet**.

The only difference is they are in order. So when you iterate over it, the result is in order.

```
@Override
public int compareTo(StockItem o) {
    System.out.println("Entering StockItem.compareTo");
    if(this==o){
        return 0;
    }
    if(o!=null){
        return this.name.compareTo(o.getName());
    }
    throw new NullPointerException();
}
```

```
public int addStock(StockItem item){
    if(item!=null){
        StockItem instock = mylist.getDefault(item.getName(),item);
    }
}
```

Above it will get the item if it already exist in the map and if it doesnt exist in the map it gonna use this item that we pass as the second argument.

Source code by Me

```
public int addStock(StockItem item){
    if(item!=null){
        // check if already have quantities of this item
        StockItem instock = mylist.getOrDefault(item.getName(),item);
        // if there are already stocks on this item, adjust the quantity.
        if(instock!=item){ // referential equality check
            item.adjustStock(instock.quantityInStock());
        }
        // On found object from Map, could be default
        mylist.put(item.getName(),item);
        return item.quantityInStock();
    }
    return 0;
}
```

Source code by Tim.

```
public int addStock(StockItem item) {
    if(item != null) {
        // check if already have quantities of this item
        StockItem inStock = list.getOrDefault(item.getName(), item);
        // If there are already stocks on this item, adjust the quantity
        if(inStock != item) {
            item.adjustStock(inStock.getQuantityStock());
        }

        list.put(item.getName(), item);
        return item.getQuantityStock();
    }
    return 0;
}
```

Above using the addMethod we are updating if exist or creating new if it doesnt.

Lec 173 Time 1:57:

In the adventure games we were returning a shallow copy(the exit). Cuz we dont want them to change. It doesnt harm anything because the caller was working with a copy its probably bit unkind to let the code think it could make changes when it actually have no effect.

So the collection framework provides a wrapper around the list,map,set collection to provide an un-modifiable collection.

Returns an **unmodifiable view** of the specified map. Query operations on the returned map "read through" to the specified map, and attempts to modify the returned map, whether direct or via its collection views, result in an **UnsupportedOperationException**.

The returned map will be serializable if the specified map is serializable.

Params: m – the map for which an unmodifiable view is to be returned.

Type parameters: <K> – the class of the map keys

<V> – the class of the map values

Returns: an unmodifiable view of the specified map.

```
@NotNull @UnmodifiableView
public static <K,V> Map<K,V> unmodifiableMap( @NotNull Map<? exte
    return new UnmodifiableMap<>(m);
}
```

Tip: This is faster than using a shallow copy.

```
public Map<String,StockItem> Items(){
    return Collections.unmodifiableMap(mylist);
}
```

Tip: for user in create cart, he uses the cart object to be stored obviously its a read only reference. Not the name cuz it will then have to searched product list.

```
private final Map<StockItem, Integer> mylist;

public int addToBasket2(StockItem key, Integer defaultValue) {
    if(key!=null && quantity>=0){
        StockItem z= mylist.getOrDefault(key,defaultValue);
    }
}
```

Above, the default value and the key must be of same Class.

Similar ways to achieve desired output: [Map Entry and Set](#)

```
public void check1(){
    System.out.println(mylist);
    Set<Map.Entry<String,StockItem>> x=mylist.entrySet();

    System.out.println(x);
    Iterator y= x.iterator();
    System.out.print("dOES IT WORK\n");
    while(y.hasNext()){

        System.out.println(y.next());
    }
}
```

```
public String toString() {
    String s="\nStock List\n";
    double totalCost=0.0;
    System.out.println("b4"+mylist);

    System.out.println("47"+mylist.entrySet());
    for(Map.Entry<String,StockItem> item : mylist.entrySet()){
        StockItem stockItem = item.getValue(); // thats gonna re
        System.out.println(52+""+item.getValue());
        double itemValue = stockItem.getPrice()*stockItem.quantiti
        s=s + stockItem+".There are"+ stockItem.quantityInStock(
        s=s + String.format("%.2f",itemValue)+"\n";
        totalCost+=itemValue;
    }
    return s+" Total stock value "+ totalCost;
}
```

```
public StockList(){
    this.mylist = new LinkedHashMap<>();
}
```

Left, maintains the order.

```
        System.out.println("price map: " + priceMap);

        Set<String> keys = priceMap.keySet();
        Collection<Integer> values = priceMap.values();
        Set<Entry<String, Integer>> entries = priceMap.entrySet();

        System.out.println("keys of Map : " + keys);
        System.out.println("values from Map :" + values);
        System.out.println("entries from Map :" + entries);

    }

}
```

Output:

```
price map: {Car=20000, Phone=200, Bike=6000, Furniture=700, TV=500}
keys of Map : [Car, Phone, Bike, Furniture, TV]
values from Map :[20000, 200, 6000, 700, 500]
entries from Map :[Car=20000, Phone=200, Bike=6000, Furniture=700,
```

Tree Map and Unmodifiable Maps

Very Cool

```
@Override
public String toString() {
    String s="\nShopping basket"+ name+" contains"+mylist.size()+
        +mylist.size()+ (mylist.size()==1? "item": "items")+
        "items\n";
    double totalCost=0;
    for(Map.Entry<StockItem, Integer> item: mylist.entrySet()){
        s=s+item.getKey()+" "+ item.getValue()+" purchase\n";
        totalCost+=item.getKey().getPrice() * item.getValue();
    }
    return s+ "Total cost"+ totalCost;
}
```

```
public class Basket {
    private final String name;
    private final Map<StockItem, Integer> mylist;

    public Basket(String name){
        this.name=name;
        this.mylist = new TreeMap<>();
    }
}
```

Here, the comparable interface is implemented in StockItem, obviously.

Note:

When make changes

```
public Map<String, StockItem> Items(){
    return Collections.unmodifiableMap(mylist);
}
```

```
Exception in thread "main" java.lang.UnsupportedOperationException Crea
    at java.base/java.util.Collections$UnmodifiableMap.put(Collections
    at Main.main(Main.java:57)
```

Note:

```
temp = new StockItem( name: "towel", price: 2.40, quantityStock: 80);  
stockList.addStock(temp);  
  
stockList.Items().get("towel").adjustStock( quantity: 1000);|
```

```
towel : price 2.4. There are 1001 in stock. Value of items: 2402.40
```

So from

previous note, and current note although items can't be added or removed. Here, the collection itself is unmodifiable. The individual objects in the collection can be accessed and modified.

Exactly what I think of shallow copy. But in unmodifiablemap here it returns an error when we do other than read .

```
stockList.get("bread").adjustStock( quantity: -100);
```

 This also works.

That can be fixed by changing the get method.

So, is it a problem. It depends.

If u design a system that the StockList Class didn't have a get method though, then you don't want the code to use get or unmodifiablemap.

So it's the collection itself which is unmodifiable and not the collection itself.

By providing shallowCopy or unmodifiable map, you can get individual items.

So if u don't want this to happen, consider some other way.

So possible option: In **Lec 175** time **18:33**

```
public Map<String , Double> priceList(){  
    Map<String, Double> prices= new LinkedHashMap<>();  
    for(Map.Entry<String,StockItem> item: mylist.entrySet()){  
        prices.put(item.getKey(),item.getValue().getPrice());  
    }  
    return Collections.unmodifiableMap(prices);|  
}
```

He created a deep copy and return a unmodifiable object.

So here not only the Map collection cant be modified not even the individual entries. Cuz its happening on a copy.

```
Set<Map.Entry<String,Double>> g1= stockList.priceList().entrySet();  
for(Map.Entry<String,Double> x:g1){  
}
```

This makes sense:

for e in ["sahil","kazim"]:

print(e)

Just like

In java, Here there is no Dyamic typing. So **Map.Entry<String,Double>**

Challenge Part 1:

Challenge Lec 178:

```
public void clearBasket(){  
    this.myList.clear();  
}
```

E.g 1:

```
public static int removeItem(Basket basket, String item, int quantity){  
    // retrieve the item from stock list  
  
    StockItem stockItem = stockList.get(item);  
    if(stockItem==null){  
        System.out.println("We dont sell "+item);  
        return 0;  
    }  
    if(basket.removeFromBasket(stockItem,quantity)==quantity){  
        return stockList.unreserveStock(item,quantity);  
    }  
    return 0;  
}
```

Here different methods are assembled.

For e.g

basket.removeFromBasket Basically remove items from basket/cart.(Basket Class)

Basket class : id, Map<String,Integer>

Which Brings it back to the store and its now unreserved.(StockList Class)

StockList Class: Map<String, StockItem>

Which calls adjustStock Method which increases the stock of the product.
(StockItem)

name,

quantityInStock,

price,

reserved (Optional) (Incase of Actual store u visiting).

Another e.g

```
stockList.Items().get("car").adjustStock( quantity: -2000);
```

Here, is a unmodifiable collection where Class fields can be modified.

Here, what if there was no car object in the list. It will give an error so do a null test first.

