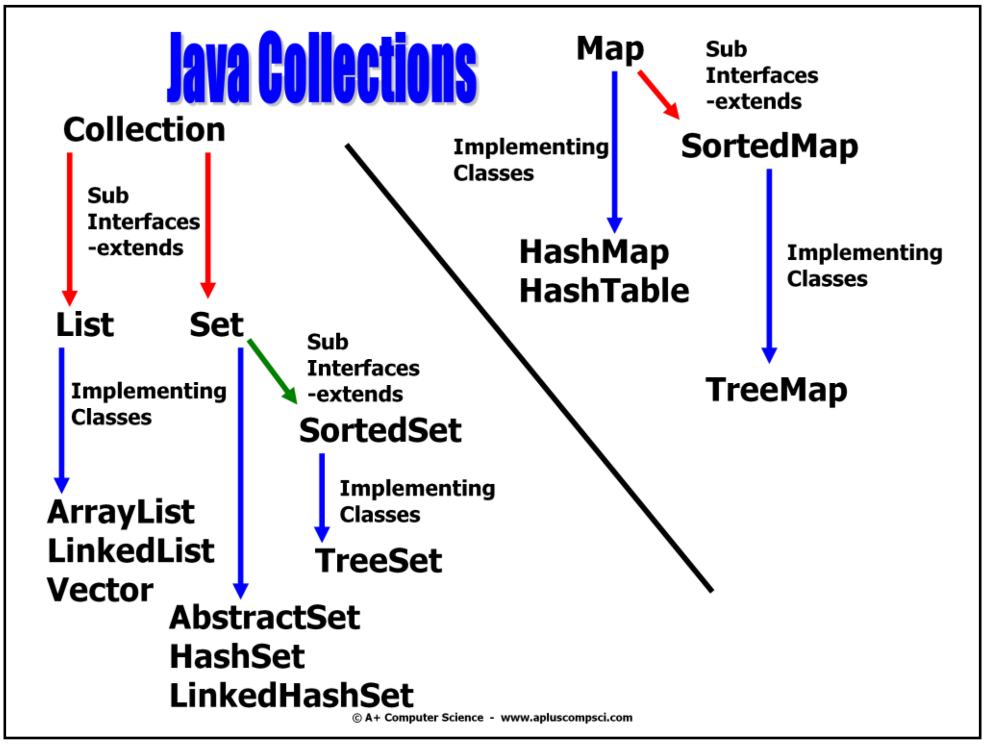
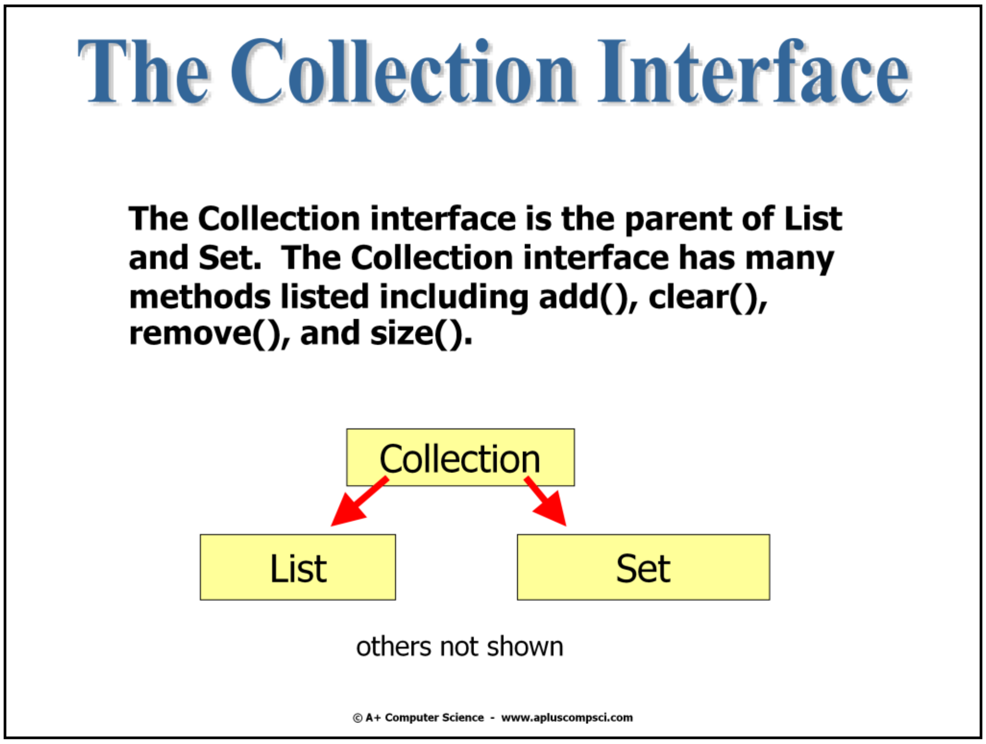
Sets

Java Interfaces

The following are important interfaces included in the Java language ::

Collection
Set
Map


Java Collections
Collection
Sub Interfaces -extends
List
Set
Implementing Classes
ArrayList
LinkedList
Vector
Sub Interfaces -extends
SortedSet
Implementing Classes
TreeSet
AbstractSet
HashSet
LinkedHashSet
Map
Sub Interfaces -extends
SortedMap
Implementing Classes
HashMap
HashTable
Implementing Classes
TreeMap
© A+ Computer Science - www.apluscompsci.com



The Collection Interface

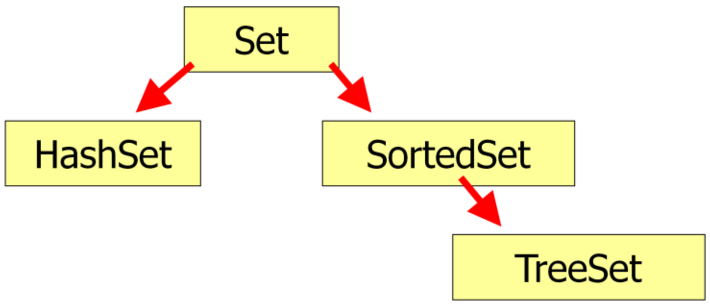
The Collection interface is the parent of List and Set. The Collection interface has many methods listed including add(), clear(), remove(), and size().

$$\begin{tikzcd} & \boxed{\text{Collection}} \arrow[dl] \arrow[dr] & \\ \boxed{\text{List}} & & \boxed{\text{Set}} \end{tikzcd}$$

others not shown

The Set Interface

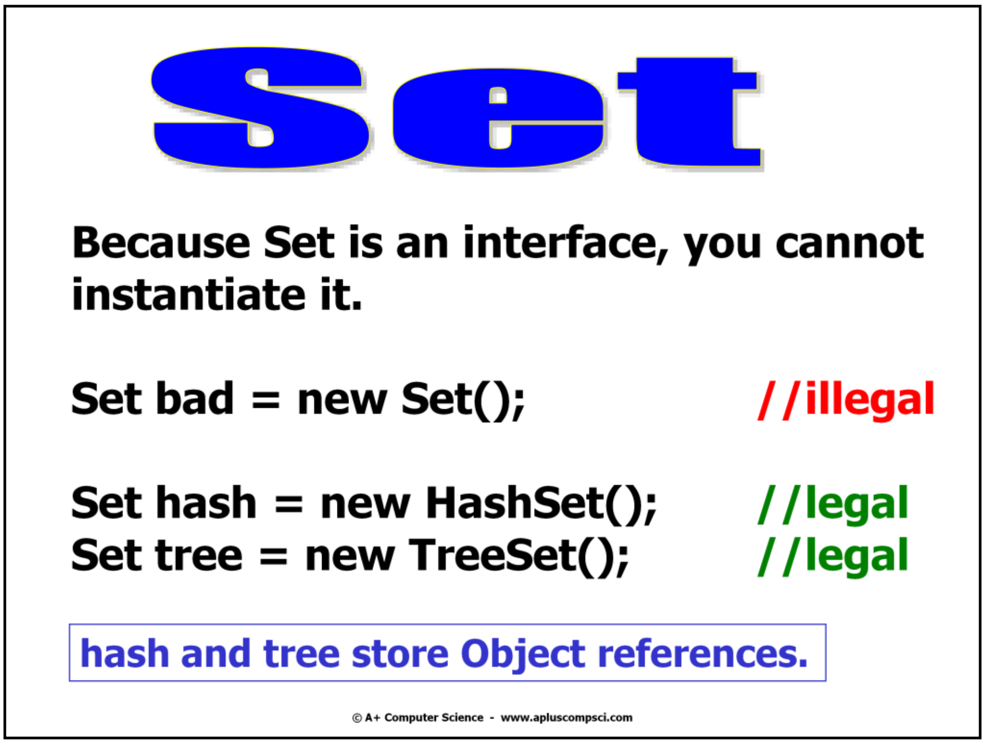
The Set interface extends the Collection interface.



Set

A set is a group of items all of the same type of which none are duplicates.

A set is very similar to an ArrayList.



Set

Because Set is an interface, you cannot instantiate it.

```
Set bad = new Set();              //illegal

Set hash = new HashSet();         //legal
Set tree = new TreeSet();         //legal
```

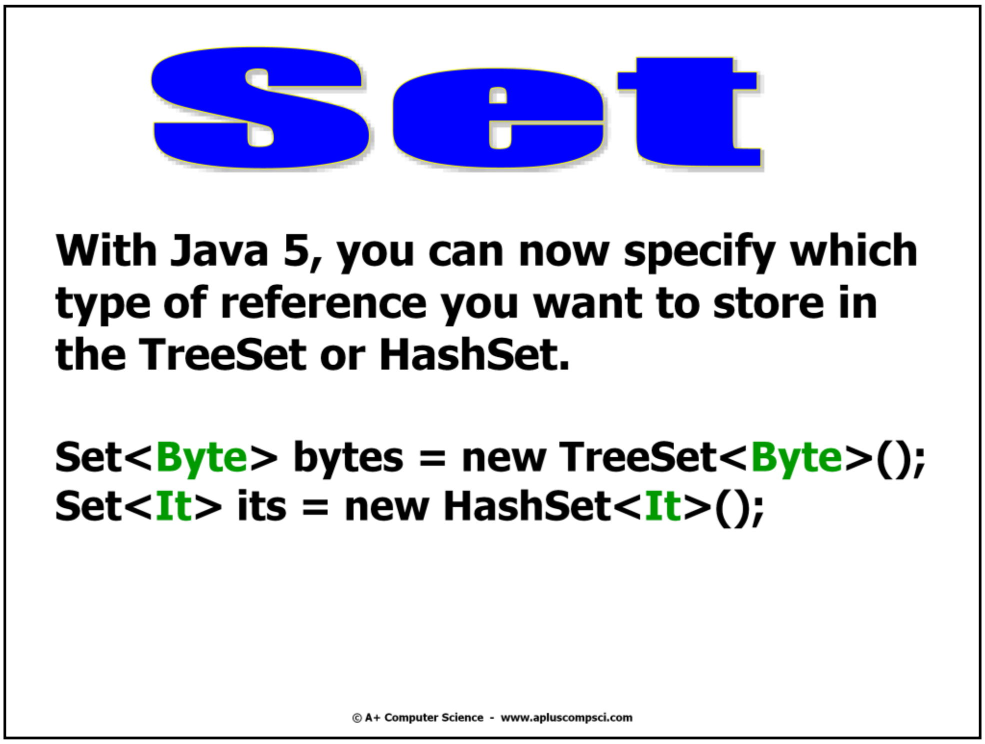
hash and tree store Object references.

Interfaces cannot be instantiated.

```
Set bad = new Set();              //illegal
```

Set can be used as a reference to one of its children.

```
Set hash = new HashSet();
Set tree = new TreeSet();
```



Set

With Java 5, you can now specify which type of reference you want to store in the TreeSet or HashSet.

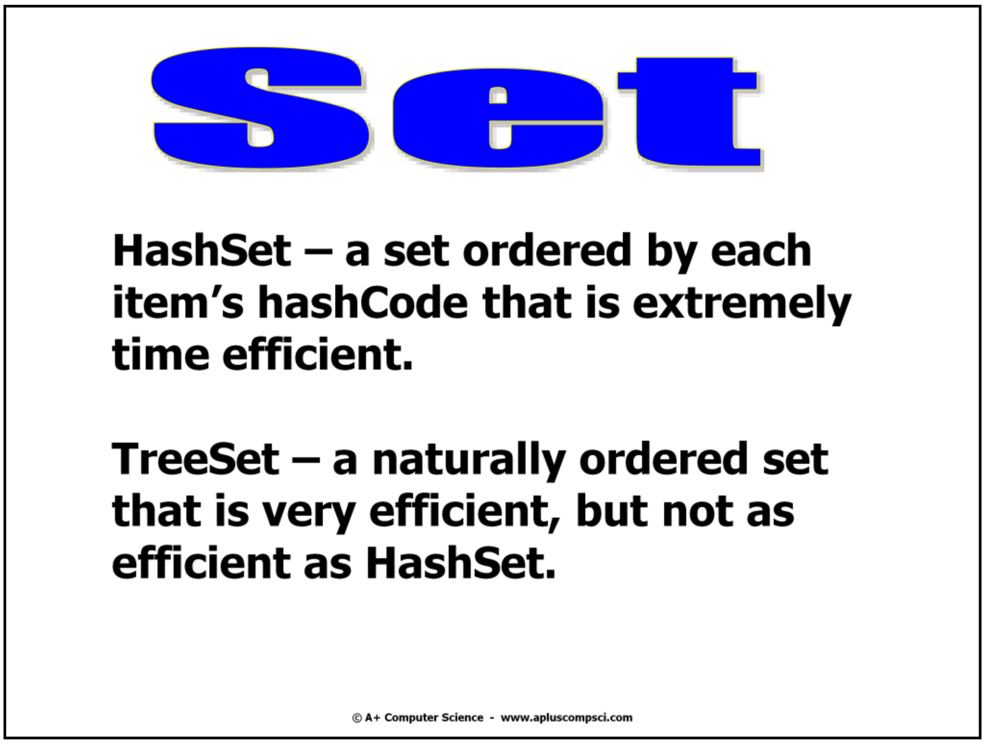
```
Set<Byte> bytes = new TreeSet<Byte>();
Set<It> its = new HashSet<It>();
```

Interfaces cannot be instantiated.

```
Set<String> bad = new Set<String>();
                                          //illegal
```

Set can be used as a reference to one of its children.

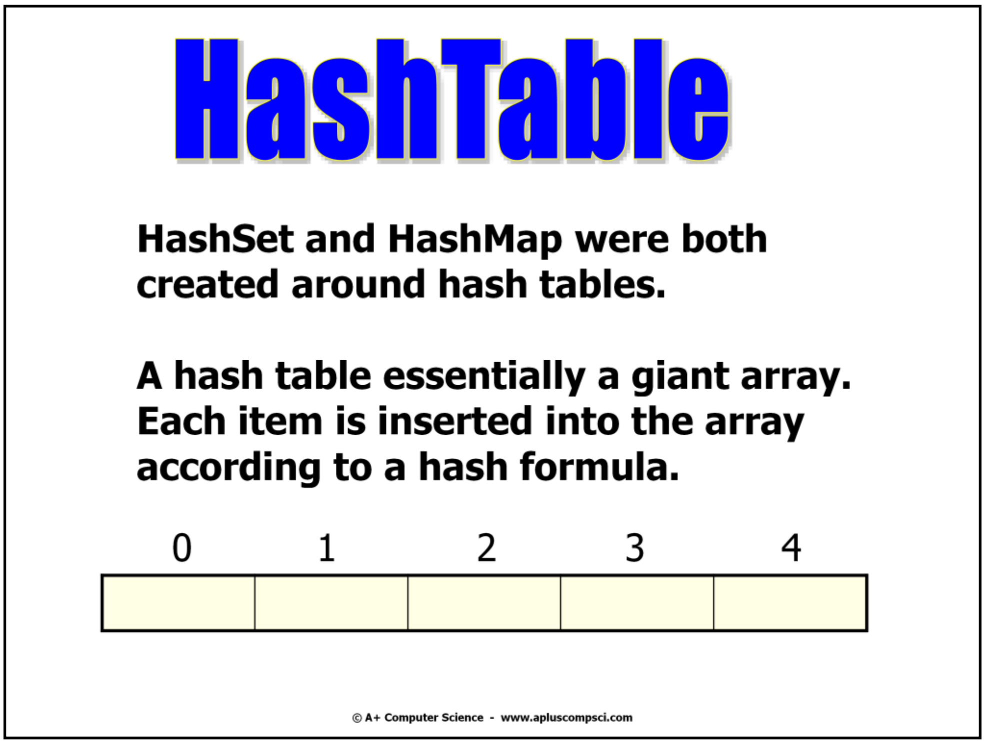
```
Set<Byte> hash = new HashSet<Byte>();
Set<String> tree = new TreeSet<String>();
```



Set

HashSet – a set ordered by each item’s hashCode that is extremely time efficient.

TreeSet – a naturally ordered set that is very efficient, but not as efficient as HashSet.



HashTable

HashSet and HashMap were both created around hash tables.

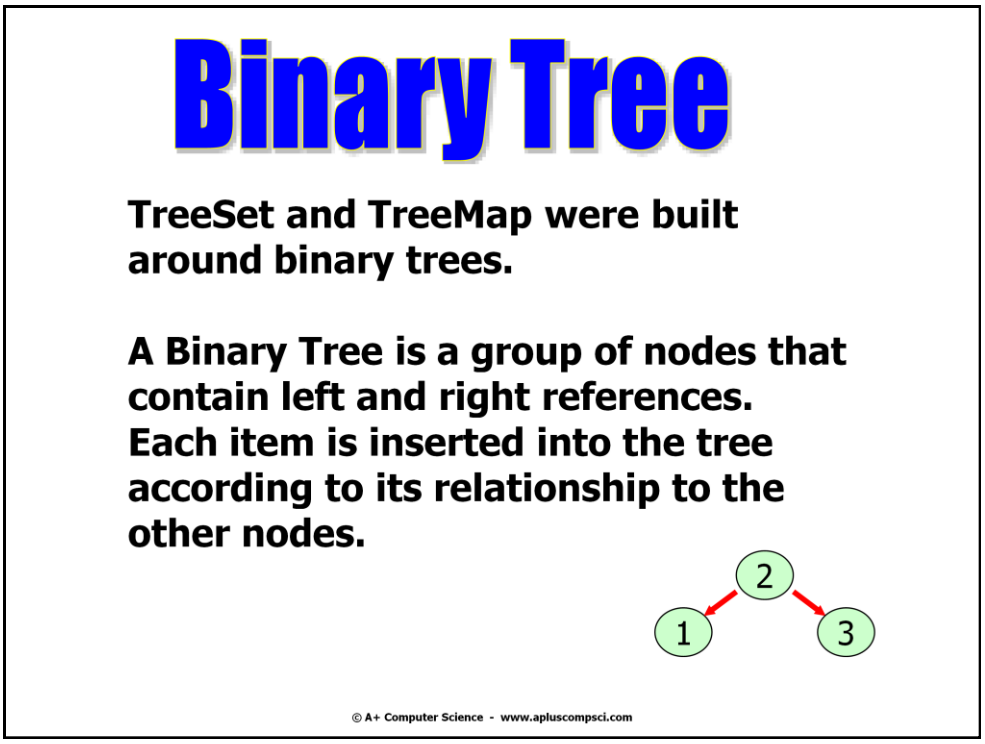
A hash table essentially a giant array. Each item is inserted into the array according to a hash formula.

Hash tables will be explained in great detail later, but the following overview will be enough for now.

Hash tables typically exist in one of two forms.

The first form is a giant array. The drawback with a giant array is that you may have wasted positions in the array where no values are being stored. The other drawback involves determining how big to make the array and when and how to increase the size of the array if needed.

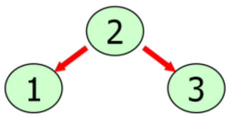
The second and probably more common form involves an array of a predetermined size where items with the same hash value are chained together. The drawback with this form is that many values could have the same hash value causing one position in the array to store most of the values.



Binary Tree

TreeSet and TreeMap were built around binary trees.

A Binary Tree is a group of nodes that contain left and right references. Each item is inserted into the tree according to its relationship to the other nodes.



Binary trees will be explained in great detail later, but the following overview will be enough for now.

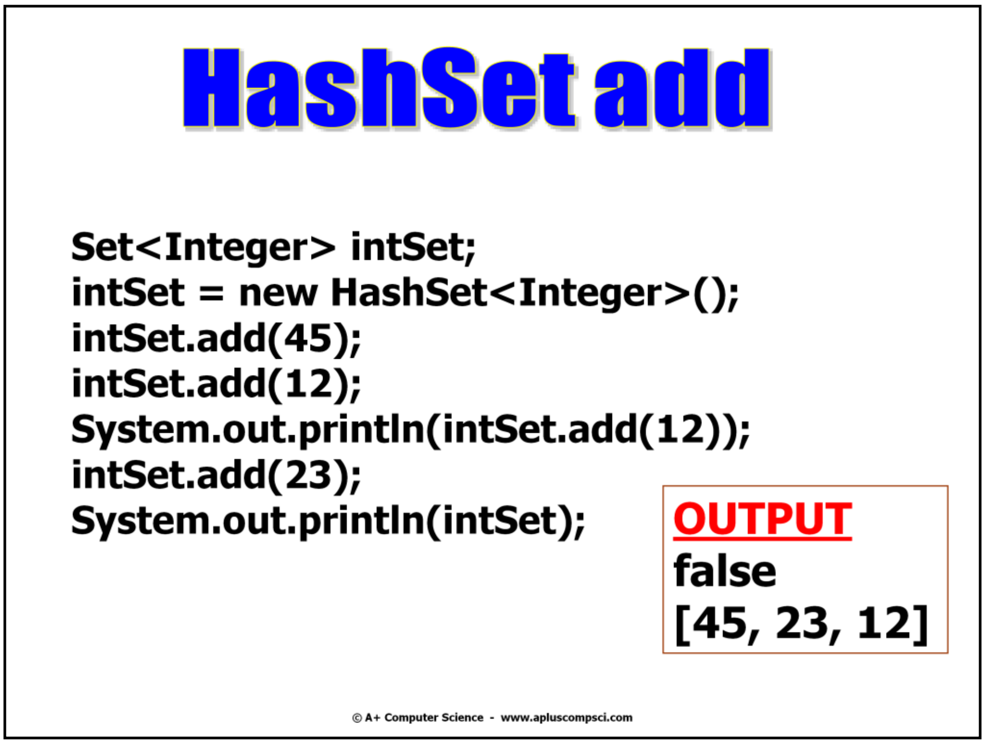
Binary Trees are organized data structures that compare new values to existing values to determine placement of the new value.

By rule, smaller items go on the left of the tree and larger items go on the right.

Set Methods

Set
frequently used methods

Name	Use
add(x)	adds item x to the set
remove(x)	removes an item from the set
clear()	removes all items from the set
size()	returns the # of items in the set



HashSet add

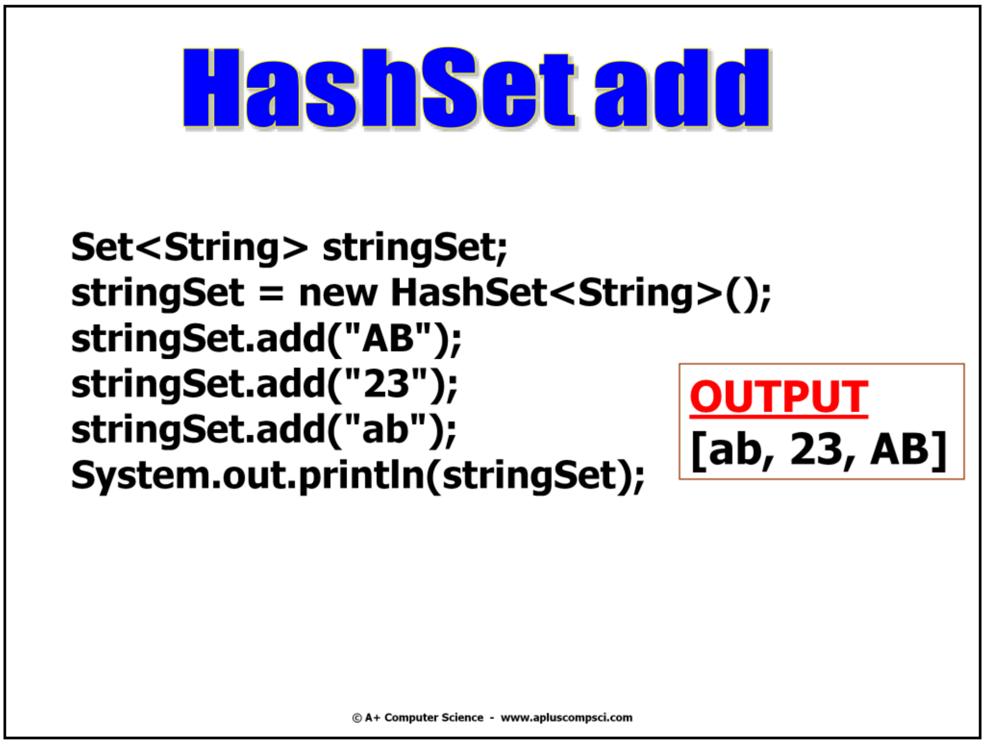
```
Set<Integer> intSet;
intSet = new HashSet<Integer>();
intSet.add(45);
intSet.add(12);
System.out.println(intSet.add(12));
intSet.add(23);
System.out.println(intSet);
```

OUTPUT

```
false
[45, 23, 12]
```

The `add()` method for `HashSet` and `TreeSet` is a return method. The reason `add()` is a boolean return method is due to the fact that an `add()` call may not add anything.

Sets do not store duplicate values so an attempt to add an item that is already present returns false. Attempting to add an item that is not present will return true.



HashSet add

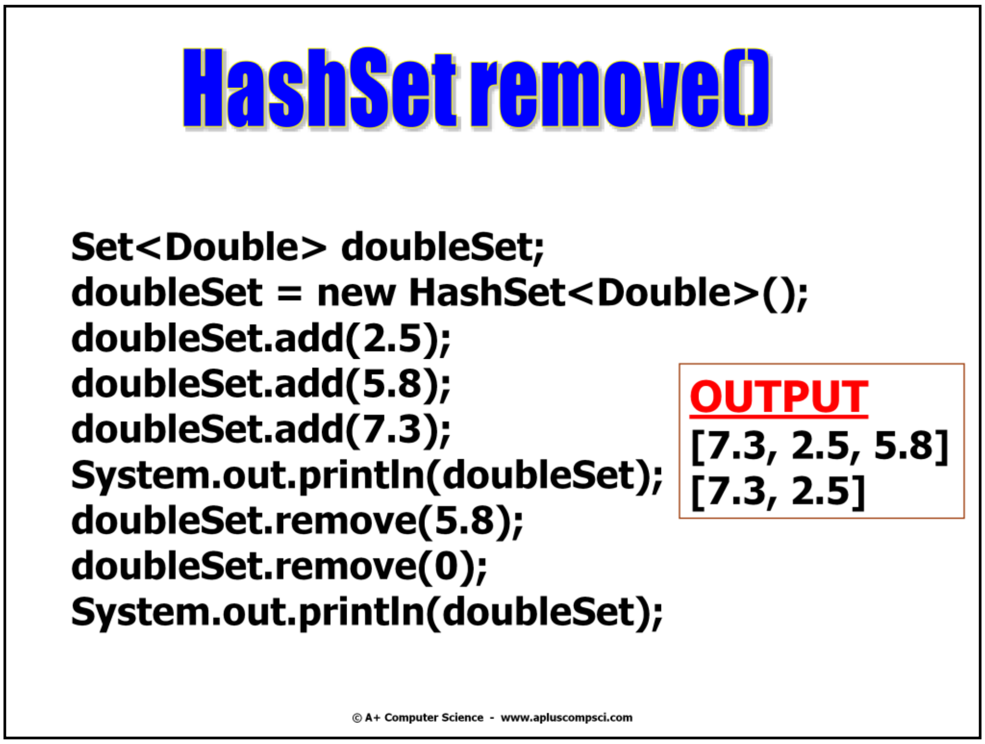
```
Set<String> stringSet;
stringSet = new HashSet<String>();
stringSet.add("AB");
stringSet.add("23");
stringSet.add("ab");
System.out.println(stringSet);
```

OUTPUT

```
[ab, 23, AB]
```

The ordering of items in a `HashSet` is not as predicable as the ordering of items in a `TreeSet`. `HashSet` orders items using a formula based on each item's `hashCode`() whereas a `TreeSet` orders items using the natural ordering of the type using the item's `compareTo().`

Open hashsetint.java hashsetstring.java



HashSet remove()

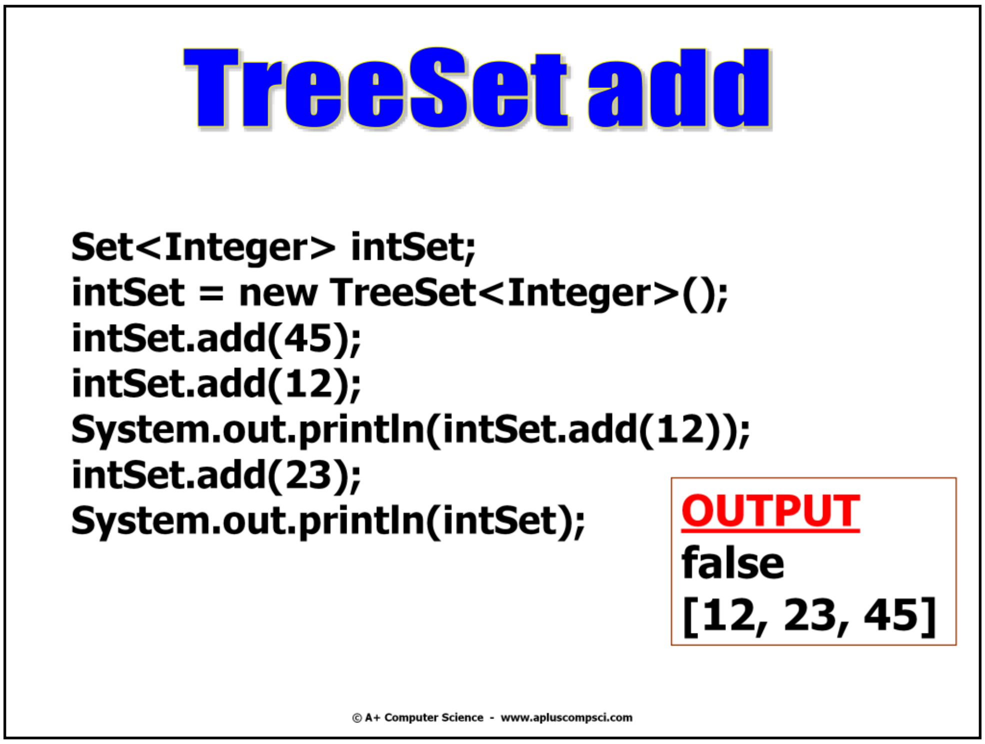
```
Set<Double> doubleSet;
doubleSet = new HashSet<Double>();
doubleSet.add(2.5);
doubleSet.add(5.8);
doubleSet.add(7.3);
System.out.println(doubleSet);
doubleSet.remove(5.8);
doubleSet.remove(0);
System.out.println(doubleSet);
```

OUTPUT

```
[7.3, 2.5, 5.8]
[7.3, 2.5]
```

The `remove()` method will remove a specified item if that item is present. If the specified item is not present, `remove()` simply does nothing.

Open hashsetremove.java



TreeSet add

```
Set<Integer> intSet;
intSet = new TreeSet<Integer>();
intSet.add(45);
intSet.add(12);
System.out.println(intSet.add(12));
intSet.add(23);
System.out.println(intSet);
```

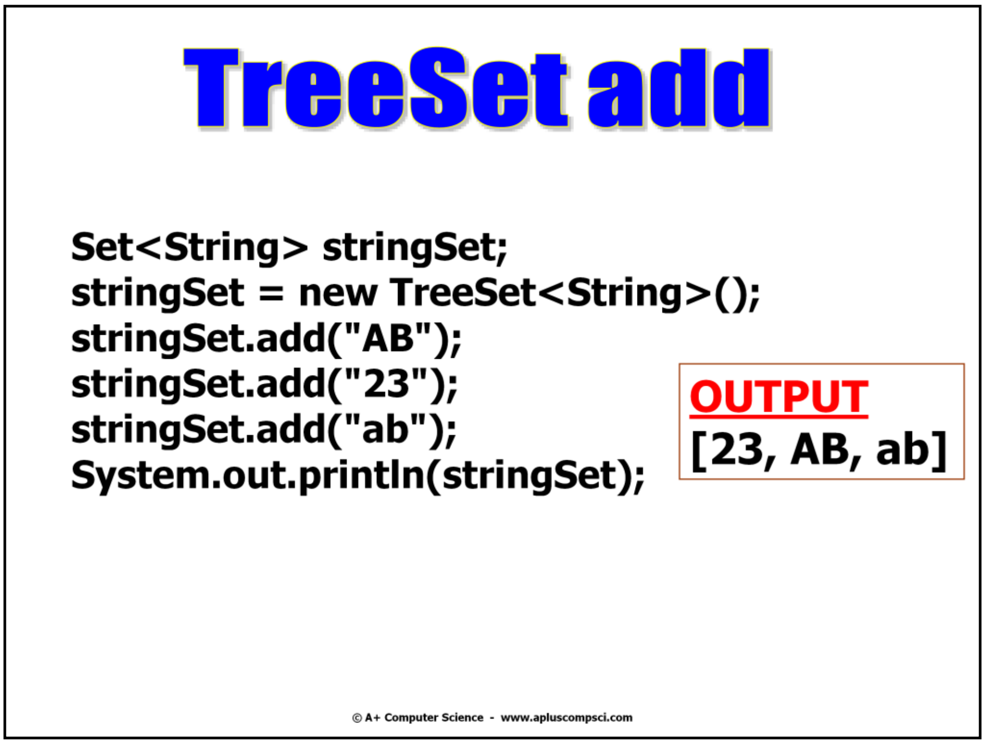
OUTPUT

```
false
[12, 23, 45]
```

© A+ Computer Science - www.apluscompsci.com

The `add()` method for `HashSet` and `TreeSet` is a return method. The reason `add()` is a boolean return method is due to the fact that an `add()` call may not add anything.

Sets do not store duplicate values so an attempt to add an item that is already present returns false. Attempting to add an item that is not present will return true.



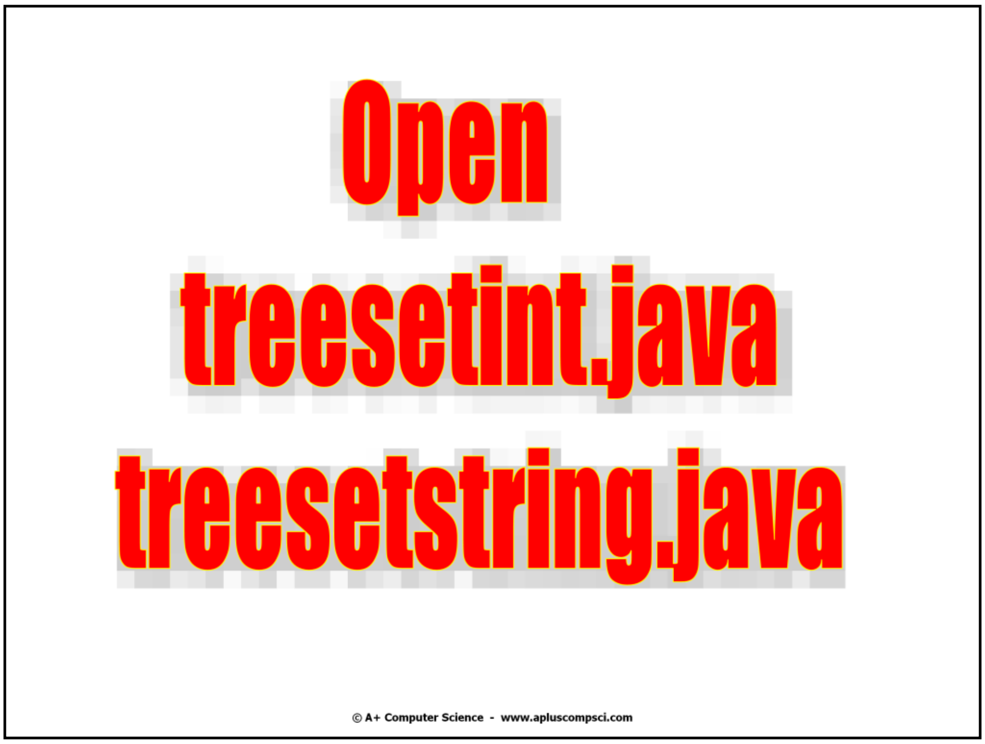
TreeSet add

```
Set<String> stringSet;
stringSet = new TreeSet<String>();
stringSet.add("AB");
stringSet.add("23");
stringSet.add("ab");
System.out.println(stringSet);
```

OUTPUT

```
[23, AB, ab]
```

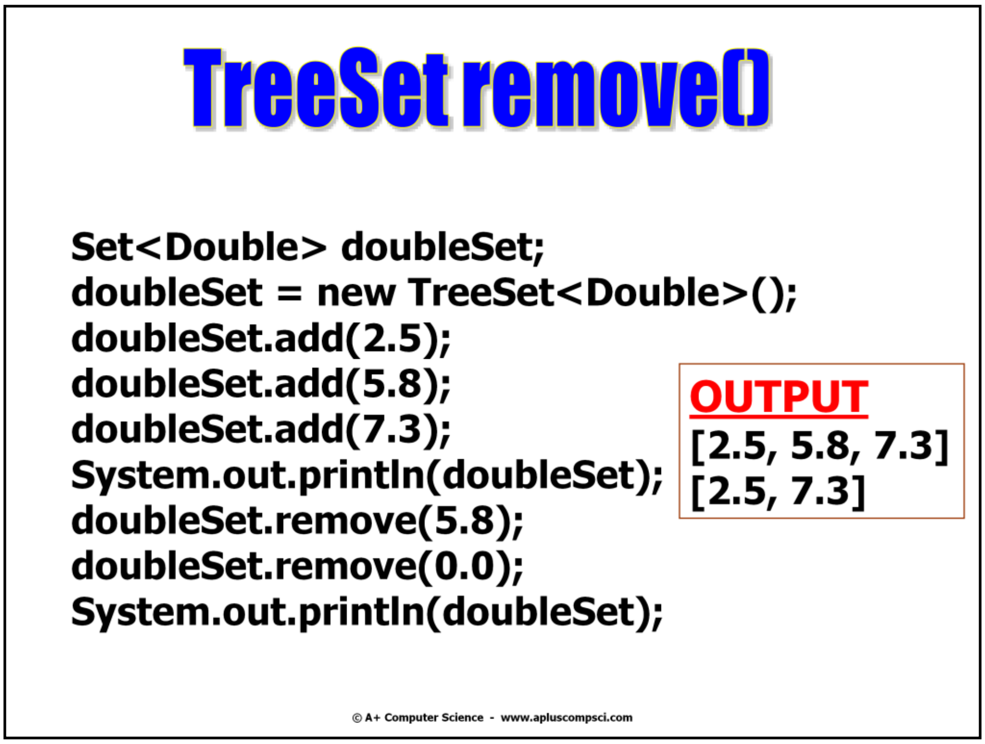
The ordering of items in a `HashSet` is not as predicable as the ordering of items in a `TreeSet`. `HashSet` orders items using a formula based on each item's `hashCode`() whereas a `TreeSet` orders items using the natural ordering of the type using the item's `compareTo().`



Open

treesetint.java

treesetstring.java



TreeSet remove()

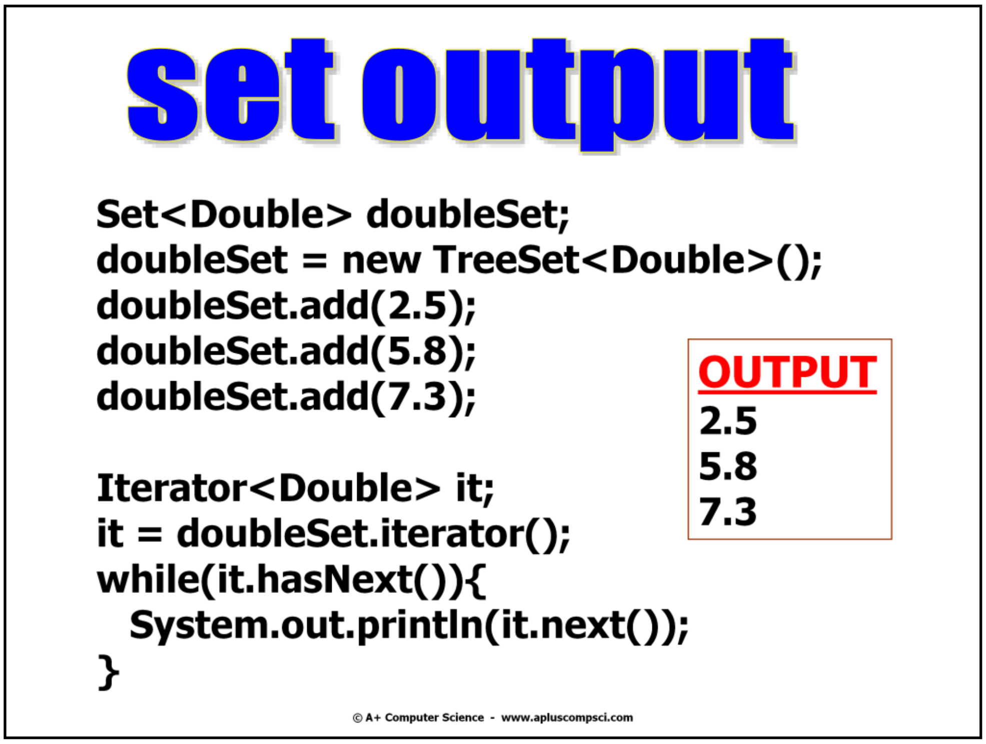
```
Set<Double> doubleSet;
doubleSet = new TreeSet<Double>();
doubleSet.add(2.5);
doubleSet.add(5.8);
doubleSet.add(7.3);
System.out.println(doubleSet);
doubleSet.remove(5.8);
doubleSet.remove(0.0);
System.out.println(doubleSet);
```

OUTPUT

```
[2.5, 5.8, 7.3]
[2.5, 7.3]
```

The `remove()` method will remove a specified item if that item is present. If the specified item is not present, `remove()` simply does nothing.

Open treesetremove.java



set output

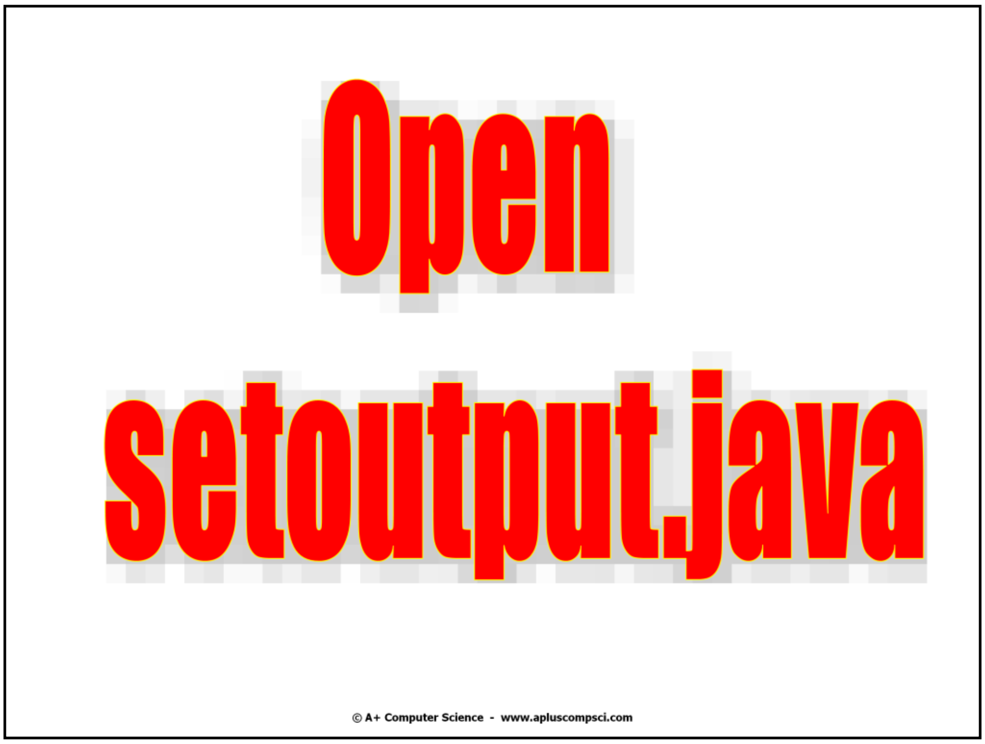
```
Set<Double> doubleSet;
doubleSet = new TreeSet<Double>();
doubleSet.add(2.5);
doubleSet.add(5.8);
doubleSet.add(7.3);

Iterator<Double> it;
it = doubleSet.iterator();
while(it.hasNext()){
  System.out.println(it.next());
}
```

OUTPUT

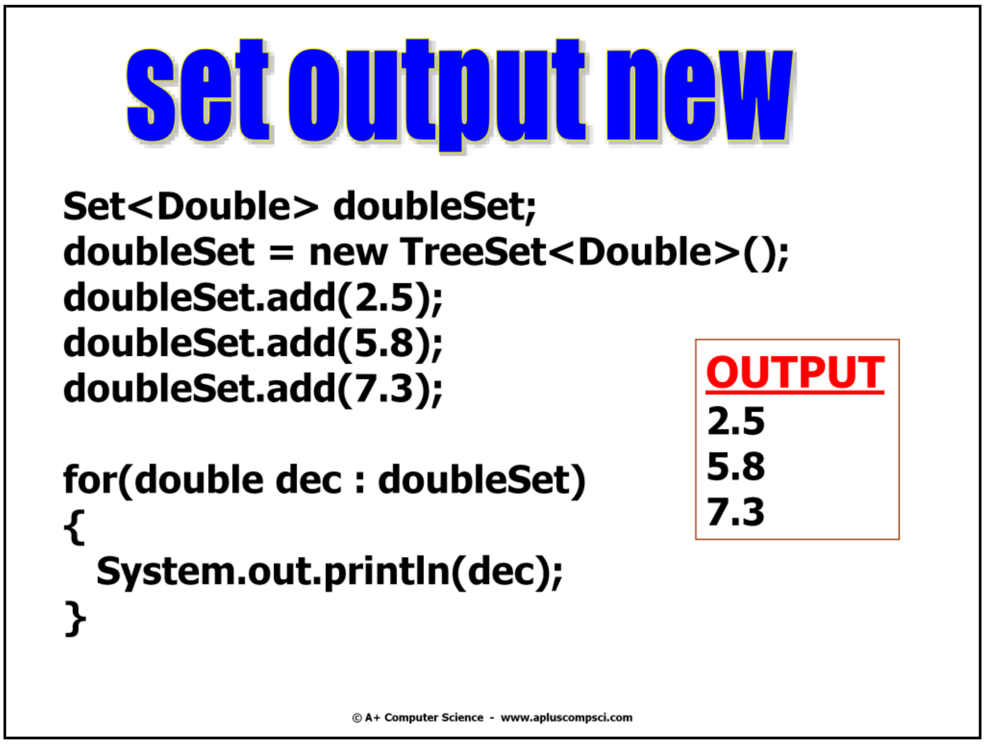
```
2.5
5.8
7.3
```

The references stored in a set are not necessarily in sequential order. The references may be all over the place in memory.

Iterators are used to access the references in a data structure in a standard way. Iterators access the references in the order specified by the data structure.



Open setoutput.java



set output new

```
Set<Double> doubleSet;
doubleSet = new TreeSet<Double>();
doubleSet.add(2.5);
doubleSet.add(5.8);
doubleSet.add(7.3);

for(double dec : doubleSet)
{
  System.out.println(dec);
}
```

OUTPUT

```
2.5
5.8
7.3
```

The new for loop iterates over all of the references in the set.

The new for loop is a little easier to code and a little easier to read than the standard iterator code needed to do the same job.

for(double dec : doubleSet)   //will iterate as long as the set contains values

//will copy the next value from doubleSet to dec each time the loop iterates

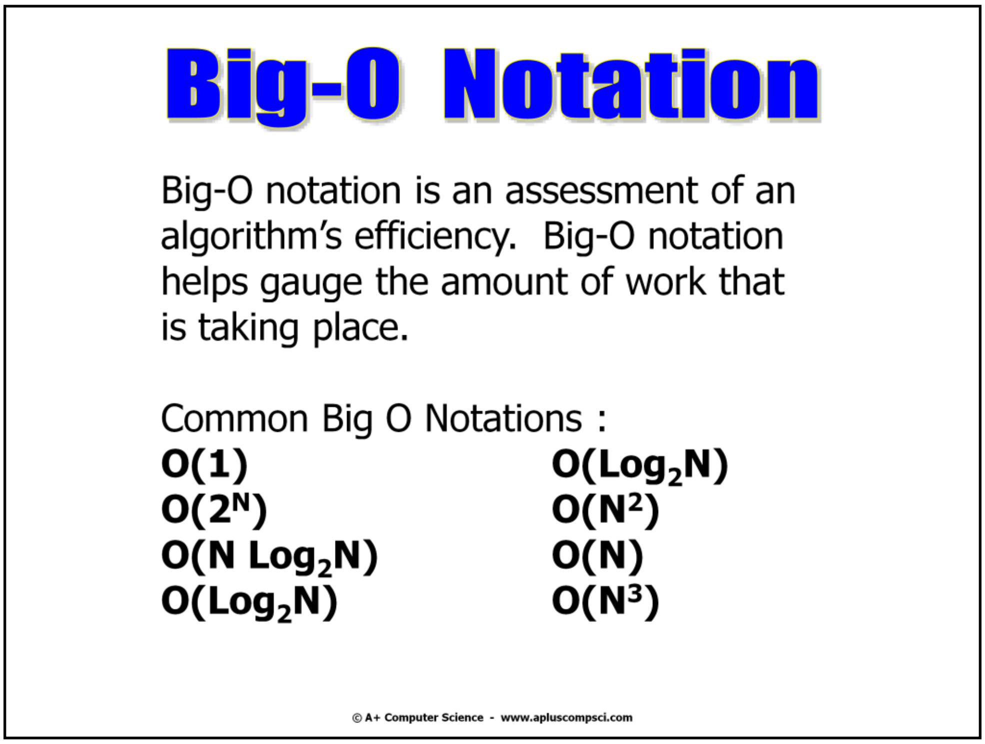
{

  //do something

}

Open setoutputnew.java setsplit.java

BigO

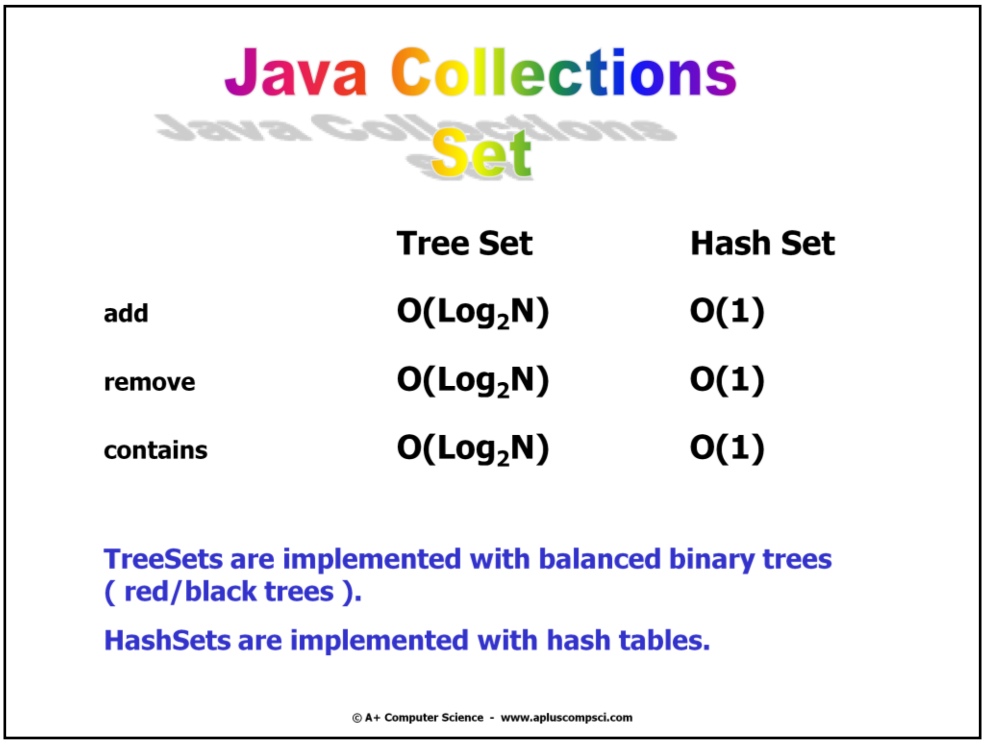


Big-O Notation

Big-O notation is an assessment of an algorithm's efficiency. Big-O notation helps gauge the amount of work that is taking place.

Common Big O Notations :

$O(1)$ **$O(Log_2N)$**
$O(2^N)$ **$O(N^2)$**
$O(N\ Log_2N)$ **$O(N)$**
$O(Log_2N)$ **$O(N^3)$**



Java Collections

Set

	Tree Set	Hash Set
add	O(Log_2N)	O(1)
remove	O(Log_2N)	O(1)
contains	O(Log_2N)	O(1)

TreeSets are implemented with balanced binary trees (red/black trees).

HashSets are implemented with hash tables.

Start work on the labs