Math Library	
Math is a final class in java	Which means it cannot be inherited
Math.abs(-1)	1 //returns the absolute value
Math.ceil(-1.9)	returns: -1.0 //Returns the smallest integer that is greater than or equal to the specified number.
Math.floor(-1.9)	returns: -2.0 // Returns the largest integer that is less than or equal to the specified number.
Math.max(1, 2)	Returns the larger of the two specified numbers.
Math.min(1, 2)	returns: 1
Math.pow(2, 3)	returns: 8.0, Returns the value of the first argument raised to the power of the second argument.
Math.random()	returns: 0.0 <= x < 1.0, Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
Math.round(1.1)	returns: 1, Returns the closest long or int, as indicated by the method's return type, to the argument
Math.round(1.5)	returns: 2,
Math.sqrt(4)	returns: 2.0, Returns the correctly rounded positive square root of a double value.

Treeset	
Set<Integer> treeSet = new TreeSet<Integer>();	Set interface does not have ceiling() method
TreeSet<Integer> treeSet = new TreeSet<Integer>();	if we use Set interface rather than TreeSet class, we cannot use ceiling() method
treeSet.add(10);	adds the element to the treeset
treeSet2.addAll(treeSet);	
System.out.println("treeSet2 after adding elements" + treeSet2);	print treeeSet
treeSet.remove(20);	removes the specific element from the treeset
for (Integer item : treeSet) { System.out.println(item); }	Iterating over tree set items
Iterator<Integer> iterator = treeSet.iterator();	Iterating using iterator
Iterator<Integer> descendingIterator = treeSet.descendingIterator();	returns an iterator over the elements in this set in descending order.
int value1 = treeSet.ceiling(25);	returns the least element in this set greater than or equal to the given element
int value2 = treeSet.floor(25);	returns the greatest element in this set less than or equal to the given element

Treeset (cont)	
Set<Integer> descendingSet = treeSet.descendingSet();	returns a reverse order view of the elements contained in this set.
Set<Integer> headSet = treeSet.headSet(30);	returns a view of the portion of this set whose elements are strictly less than toElement.
Set<Integer> headSet2 = treeSet.headSet(30, true);	returns a view of the portion of this set whose elements are less than or equal to toElement.
Set<Integer> tailSet = treeSet.tailSet(30);	returns a view of the portion of this set whose elements are greater than or equal to fromElement.
Set<Integer> tailSet2 = treeSet.tailSet(30, false);	returns a view of the portion of this set whose elements are strictly greater than fromElement.
Set<Integer> subSet = treeSet.subSet(20, 40);	returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
Set<Integer> subSet2 = treeSet.subSet(20, true, 40, true);	returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, inclusive.
Integer pollFirst = treeSet.pollFirst();	retrieves and removes the first (lowest) element, or returns null if this set is empty.
Integer pollLast = treeSet.pollLast();	retrieves and removes the last (highest) element, or returns null if this set is empty.
boolean remove = treeSet.remove(20);	returns true if this set contained the specified element
boolean isEmpty = treeSet.isEmpty();	returns true if this set contains no elements.
int size = treeSet.size();	returns the number of elements in this set (its cardinality).
boolean contains = treeSet.contains(20);	returns true if this set contains the specified element.
treeSet.clear();	

TreeSet is implemented using a tree structure(red-black tree in algorithm book)
TreeSet is ordered, sorted, and navigable
TreeSet does not allow duplicate elements
TreeSet allows null elements

Primitive Data types		
byte	Size: 1 byte	Range: -128 to 127
short	Size: 2 bytes	Range: -32,768 to 32,767
int	Size: 4 bytes	Range: -2,147,483,648 to 2,147,483,647
long	Size: 8 bytes	Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	Size: 4 bytes	Range: 3.4e−038 to 3.4e+038
double	Size: 8 bytes	Range: 1.7e−308 to 1.7e+308
boolean	Size: 1 bit	Range: true or false
char	Size: 2 bytes	Range: 0 to 65,536

Operators	
a * b	Multiplication
a + b	
a - b	
a / b	Division
a % b	Modulo
a++	Postincrement
++a	Preincrement
b--	Postdecrement
--b	Predecrement
a operator= 3	+, -, *, /, %, <<, >>, >>>
a operator 0b1010	&, |, ^
a relationalOperator b	==, !=, <, <=, >, >=
a logicalOperator b	&&, ||,
!x	not operator
TODO: terenary	
TODO: shift operator	

loops
for (int number : numbers) { System.out.println("Element: " + number); }
for (int i = 0; i < numbers.size(); i++) { int element = numbers.get(i); System.out.println("Element at index " + i + ": " + element); }

Set

The set interface is present in java.util package	which is extended from collection interface
Interfaces extended from Set interface:	SortedSet, NavigableSet
Classes that implements Set Interface	HashSet, EnumSet, LinkedHashSet, TreeSet
SortedSet Interface	contains the methods inherited from the Set interface and adds a feature that stores all the elements in this interface to be stored in a sorted manner
Navigable Interface:	provides the implementation to navigate through the Set.
Treeset	implementation of a self-balancing tree

set is implemented using hash table
set is unordered
set does not allow duplicate elements
set allows null elements

LinkedHashSet

create a linkedHashSet of characters	Set<Character> linkedHashSet = new LinkedHashSet<Character>();
add element	linkedHashSet.add('a');
remove the element	linkedHashSet.remove('a');
check if the element is present	linkedHashSet.contains('a')
size	linkedHashSet.size()
is empty	linkedHashSet.isEmpty()
Iterator	Iterator iterator = linkedHashSet.iterator();
to Array	linkedHashSet.toArray()
to string	linkedHashSet.toString()
equals	linkedHashSet.equals(linkedHashSet)
hashCode()	linkedHashSet.hashCode()

linkedHashSet is a subclass of HashSet
linkedHashSet maintains insertion order
linkedHashSet is slower than HashSet
linkedHashSet is faster than TreeSet
when to use: when you want to maintain insertion order
when not to use: when you want to sort the elements

HashSet

Creating the hashSet	Set<String> hashSet = new HashSet<String>();
Adding element	hashSet.add("element1");
Print hashSet	System.out.println("hashset after adding elements" + hashSet);
remove element	hashSet.remove("element2");

HashSet (cont)

Iterating over hash set items	for (String item : hashSet) {System.out.println(item);}
// Iterating using iterator	Iterator<String> iterator = hashSet.iterator();
checking if hashSet contains an element	hashSet.contains("element1")
checking if hashSet is empty	hashSet.isEmpty()
checking size of hashSet	hashSet.size()
clearing hashSet	hashSet.clear();

Hashset is implemented using hash table
Hashset is unordered
Hashset does not allow duplicate elements
Hashset allows null elements

Enum hashset

create enumSet	EnumSet<Days> hashSet = EnumSet.of(Days.MONDAY, Days.TUESDAY, Days.W-EDNESDAY, Days.THURSDAY, Days.FRIDAY);
[MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY]	EnumSet<Days> workingDays = EnumSet.range(Days.MONDAY, Days.FRIDAY);
[SATURDAY, SUNDAY]	EnumSet<Days> weekEnds = EnumSet.complementOf(workingDays);
[MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]	EnumSet<Days> allDays = EnumSet.allOf(Days.class);
[]	EnumSet<Days> noDays = EnumSet.noneOf(Days.class);
add	noDays.add(Days.MONDAY);
remove	noDays.remove(Days.MONDAY);
check if element is present	noDays.contains(Days.MONDAY)
check if enum hashset are equal	weekDays.equals(noDays)
iterator	Iterator iterator = noDays.iterator();
get the size	noDays.size()
	noDays.toArray()
	noDays.clear();

when to use: when you want to use a set of enum constants
when not to use: when you want to use a set of non-enum constants