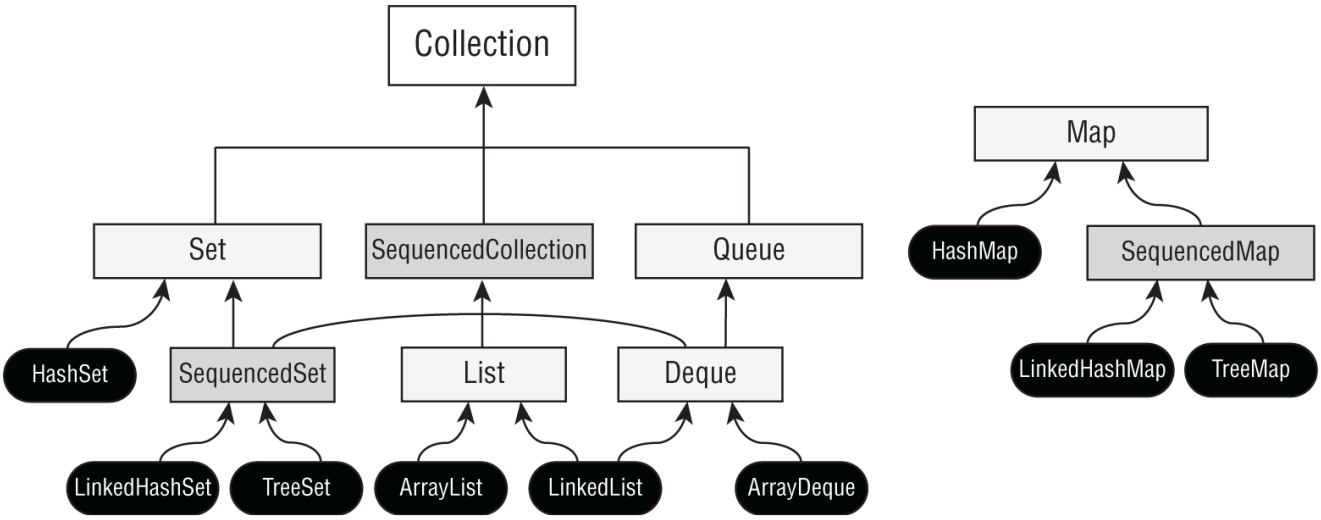
Collections and Generics

Common Collection APIs

Collection

Is a group of objects contained in a single object



Generic -> a way for parameterized type

Diamond operator <>

A shorthand notation that allows you to omit the generic type from the right side of a statement when the type can be inferred.

USING BOTH SHORTENERS

What happens if you use both `var` and the diamond operator?

```
var map = new HashMap<>();
```

Believe it or not, this does compile! If you try to have them both infer, there isn't enough information and you get `Object` as the generic type. This is equivalent to the following:

```
HashMap<Object, Object> map = new HashMap<Object, Object>();
```

.add()

Inserts a new element into the Collection and returns whether it was successful

```
public boolean add(E element)

//signature of .add() method
```

.remove()

Removes a single matching value in the Collection and return whether it was successful.

```
public boolean remove(Object object)

//signature of .remove() method
```

The `.isEmpty()` and `.size()` methods look at how many elements are in the Collection.

```
public boolean isEmpty()
public int size()
```

.clear()

Provides an easy way to discard all elements of the Collection.

```
public void clear()
```

.contains()

Checks whether a certain value is in the Collection.

```
public boolean contains()
```

.removeIf()

Removes all elements that match a condition. We can specify what should be deleted using a block of code (more specific, maybe a lambda) or even a method reference.

```
public boolean removeIf(Predicate<? super E> filter)

----------------Example-----------------------

4: Collection<String> list = new ArrayList<>();
5: list.add("Magician");
6: list.add("Assistant");
7: System.out.println(list);     // [Magician, Assistant]
```

```
8: list.removeIf(s -> s.startsWith("A"));
9: System.out.println(list);     // [Magician]
```

.forEach()

You can call on a Collection instead of writing a loop. It uses a Consumer that takes a single parameter and doesn't return anything.

```
public void forEach(Consumer<? super T> action)

--------------------Example------------------

Collection<String> cats = List.of("Annie", "Ripley");
cats.forEach(System.out::println);
cats.forEach(c -> System.out.println(c));
```

Don't forget that exists other methods for looping in a Collection:

```
//coll is a Collection object

for(String elem : coll){
    System.out.println(elem);
}
```

.equals()

There is a custom implementation of `equals()` so you can compare 2 Collection to compare the type and contents. See that implementation vary.

```
boolean equals(Object object)

------------Example--------------

23: var list1 = List.of(1, 2);
24: var list2 = List.of(2, 1);
25: var set1 = Set.of(1, 2);
26: var set2 = Set.of(2, 1);
27:
28: System.out.println(list1.equals(list2));  // false
29: System.out.println(set1.equals(set2));    // true
30: System.out.println(list1.equals(set1));   // false
```

List

> 🖊 List
>
> Is an ordered collection of elements that allows duplicate entries. Elements in a list can be accessed by an `int` index.

Implementation:

1. ArrayList - Like a resizable array. The main benefit of an ArrayList is that you can look up any element in constant time. Instead, adding and removing an element is slower than accessing an element.
2. LinkedList - Special because it implements both List and Deque. it also has all methods of a List and additional methods to facilitate adding or removing from the beginning and/or end of the list. The main benefits of LinkedList are that you can access, add to, and remove from the beginning and end of the list in constant time.

Creating List with a Factory

Method	Description	Can add elements?	Can replace elements?	Can delete elements?
`Arrays.asList(varargs)`	Returns fixed size list backed by an array	No	Yes	No
`List.of(varargs)`	Returns immutable list	No	No	No
`List.copyOf(collection)`	Returns immutable list with copy of original collection's values	No	No	No

```
16: String[] array = new String[] {"a", "b", "c"};
17: List<String> asList = Arrays.asList(array); // [a, b, c]
18: List<String> of = List.of(array);           // [a, b, c]
19: List<String> copy = List.copyOf(asList);    // [a, b, c]
20:
21: array[0] = "z";
22:
23: System.out.println(asList);                 // [z, b, c]
24: System.out.println(of);                     // [a, b, c]
25: System.out.println(copy);                   // [a, b, c]
26:
27: asList.set(0, "x");
28: System.out.println(Arrays.toString(array)); // [x, b, c]
```

Creating a List with a Constructor

For LinkedList:

```
var linked1 = new LinkedList<String>();
var linked2 = new LinkedList<String>(linked1);
```

For ArrayList:

```
var list1 = new ArrayList<String>();
var list2 = new ArrayList<String>(list1);
var list3 = new ArrayList<String>(10); //The final example says to create an
`ArrayList` containing a specific number of slots
```

Working with List methods

Method	Description
`boolean add(E element)`	Adds element to end (available on all `Collection` APIs).
`void add(int index, E element)`	Adds element at index and moves the rest toward the end.
`E get(int index)`	Returns element at index.
`int indexOf(Object o)`	Returns the index of the first matching element or -1 if not found.
`int lastIndexOf(Object o)`	Returns the index of the last matching element or -1 if not found.
`E remove(int index)`	Removes element at index and moves the rest toward the front.
`default void replaceAll(UnaryOperator<E> op)`	Replaces each element in list with the result of operator.
`E set(int index, E e)`	Replaces element at index and returns original. Throws `IndexOutOfBoundsException` if index is invalid.
`default void sort(Comparator<? super E> c)`	Sorts list. We cover this later in the chapter in the "Sorting Data" section.

Converting from List to Array

```
13: List<String> list = new ArrayList<>();
14: list.add("hawk");
15: list.add("robin");
16: Object[] objectArray = list.toArray();
17: String[] stringArray = list.toArray(new String[0]);
18: list.clear();
19: System.out.println(objectArray.length);     // 2
20: System.out.println(stringArray.length);     // 2
```

Set

> Set
>
> Is a collection that does not allow duplicate entries.

Implementation of Set:

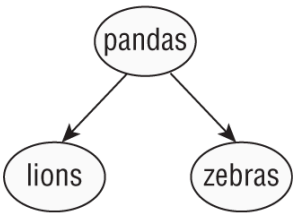
1. `HashSet` - stores its elements in a *hash table*, which means hte keys are a hash and the values are an Object. This means that the HashSet uses the hashCode() method of the objects to retrieve them more efficiently.
2. `LinkedHashSet` - basically a HashSet with an imaginary `LinkedList` running across its elements. Also includes methods to add/remove elements from the front or back of the set.
3. `TreeSet` - stores its elements in a sorted tree structure. The set is always in sorted order. The trade-off is that adding or removing an element could take longer that with a `Hashset`, especially as the tree grows larger.

HashSet

`hashCode()` Value	Data
–995544615	pandas
...	
–705903059	zebras
...	
102978519	lions

LinkedHashSet

TreeSet



Working with Set methods

You can create an immutable Set in one line or make copy of an existing one.

```
Set<Character> letters = Set.of('c', 'a', 't');
Set<Character> copy = Set.copyOf(letters);
```

```
3: Set<Integer> set = new HashSet<>();
4: boolean b1 = set.add(66);  // true
5: boolean b2 = set.add(10);  // true
6: boolean b3 = set.add(66);  // false
7: boolean b4 = set.add(8);   // true
8: for (Integer value: set)
9:    System.out.print(value + ","); // 66,8,10,
```

If we replace `Hashset` with `LinkedHashSet`

```
3: Set<Integer> set = new LinkedHashSet<>();
4: boolean b1 = set.add(66);  // true
5: boolean b2 = set.add(10);  // true
6: boolean b3 = set.add(66);  // false
7: boolean b4 = set.add(8);   // true
8: for (Integer value: set)
9:    System.out.print(value + ","); // 66,10,8,
```

If we replace with `TreeSet`:

```
3: Set<Integer> set = new TreeSet<>();
4: boolean b1 = set.add(66);  // true
5: boolean b2 = set.add(10);  // true
6: boolean b3 = set.add(66);  // false
7: boolean b4 = set.add(8);   // true
8: for (Integer value: set)
9:    System.out.print(value + ","); // 8,10,66
```

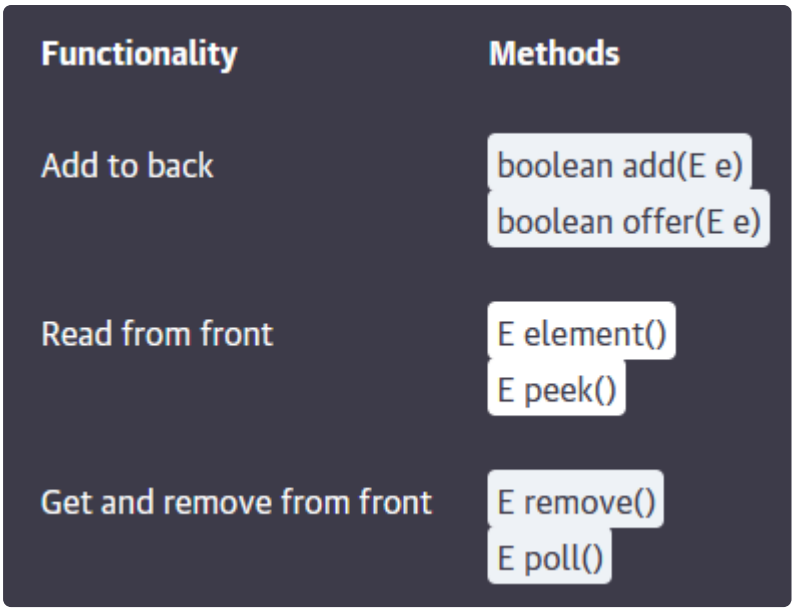
Queue and Deque

> **Queue**
>
> A collection that orders its elements in a specific order for processing.

> **Deque**
>
> A subinterface of `Queue` that allows access at both ends.

Working with Queue and Deque methods

For Queue



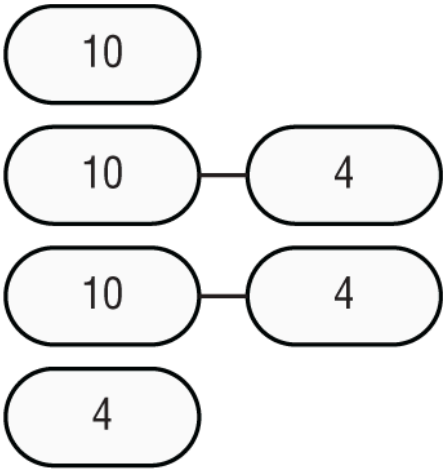

```
4: Queue<Integer> queue = new LinkedList<>();
5: queue.add(10);
6: queue.add(4);
7: System.out.println(queue.remove());   // 10
8: System.out.println(queue.peek());     // 4
```

For Deque

Functionality	Methods
Add to front	void addFirst(E e) boolean offerFirst(E e)
Add to back	void addLast(E e) public boolean offerLast(E e)
Read from front	E getFirst() E peekFirst()
Read from back	E getLast() E peekLast()
Get and remove from front	E removeFirst() E pollFirst()
Get and remove from back	E removeLast() E pollLast()

```
12: Deque<Integer> deque = new LinkedList<>();
```

```
13: deque.offerFirst(10); // true
14: deque.offerLast(4);   // true
15: deque.peekFirst();    // 10
16: deque.pollFirst();    // 10
17: deque.pollLast();     // 4
18: deque.pollFirst();    // null
19: deque.peekFirst();    // null
```

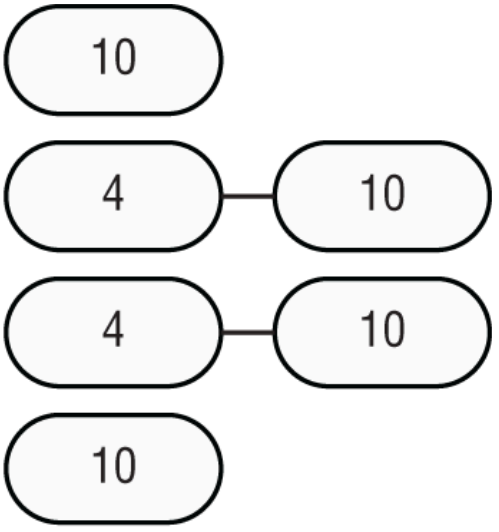


Using Deque like a Stack

Functionality	Methods
Add to the front/top	void push(E e)
Remove from the front/top	E pop()
Get first element	E peek()

```
12: Deque<Integer> stack = new ArrayDeque<>();
```

```
13: stack.push(10);
14: stack.push(4);
15: stack.peek();     // 4
16: stack.pop();      // 4
17: stack.pop();      // 10
18: stack.peek();     // null
```



> When using a `Deque`, it is really important to determine if it is being used as a FIFO queue, a LIFO stack, or a double-ended queue. To review, a FIFO queue is like a line of people. You get on in the back and off in the front. A LIFO stack is like a stack of plates. You put the plate on the top and take it off the top. A double-ended queue uses both ends.

Map

> 🖊 Map
>
> A collection that maps keys to values, with no duplicate keys allowed. The elements in a map are key/value pairs.

Just like `List` and `Set`, there is a factory method to create a `Map`. You pass up to 10 pairs of keys and values.

```
Map.of("key1", "value1", "key2", "value2");
```

Passing keys and values is harder to read because you have to keep track of which parameter is which. `Map` also provides a method that lets you supply key/value pairs.

```
Map.ofEntries(
   Map.entry("key1", "value1"),
   Map.entry("key2", "value2"));
```

Comparing Map implementation

1. `HashMap` - stores the keys in a hash table. This means that it uses the `hashCode()`
2. `LinkedHashMap` - support iterating over the elements in a well-defined order. This is generally the insertion order, although it also includes methods to add/remove elements at the front/back of the map.
3. `TreeMap` - stores the keys in a sorted tree structure. Benefits - Keys are always in sorted order. Trade-off - adding and checking whether a key is present takes longer as the tree grows larger.

Working with Map methods

Method	Description
`void clear()`	Removes all keys and values from map.
`boolean containsKey(Object key)`	Returns whether key is in map.
`boolean containsValue(Object value)`	Returns whether value is in map.
`Set<Map.Entry<K,V>> entrySet()`	Returns `Set` of key/value pairs.
`void forEach(BiConsumer<K, V> action)`	Loops through each key/value pair.
`V get(Object key)`	Returns value mapped by key or `null` if none is mapped.
`V getOrDefault(Object key, V defaultValue)`	Returns value mapped by key or default value if none is mapped.
`boolean isEmpty()`	Returns whether map is empty.
`Set<K> keySet()`	Returns set of all keys.
`V merge(K key, V value, BiFunction<V, V, V> func)`	Sets value if key not set. Runs function if key is set, to determine new value. Removes if value is `null`.
`V put(K key, V value)`	Adds or replaces key/value pair. Returns previous value or `null`.
`V putIfAbsent(K key, V value)`	Adds value if key not present and returns null. Otherwise, returns existing value.
`V remove(Object key)`	Removes and returns value mapped to key. Returns `null` if none.
`V replace(K key, V value)`	Replaces value for given key if key is set. Returns original value or `null` if none.
`void replaceAll(BiFunction<K, V, V> func)`	Replaces each value with results of function.
`int size()`	Returns number of entries (key/value pairs) in map.
`Collection<V> values()`	Returns `Collection` of all values.

Iterating

```
--------using forEach() method--------------
Map<Integer, Character> map = new HashMap<>();
map.put(1, 'a');
map.put(2, 'b');
map.put(3, 'c');
map.forEach((k, v) -> System.out.println(v));
```

```
----------using method reference-----------
map.values().forEach(System.out::println);

---------using entrySet()-------------------
map.entrySet().forEach(e ->
   System.out.println(e.getKey() + " " + e.getValue()));
```

Getting values safely

```
3: Map<Character, String> map = new HashMap<>();
4: map.put('x', "spot");
5: System.out.println("X marks the " + map.get('x'));
6: System.out.println("X marks the " + map.getOrDefault('x', "")); //safely
7: System.out.println("Y marks the " + map.get('y'));
8: System.out.println("Y marks the " + map.getOrDefault('y', "")); //safely

----------------------------------------------------------------------

X marks the spot
X marks the spot
Y marks the null
Y marks the
```

Replacing values

```
21: Map<Integer, Integer> map = new HashMap<>();
22: map.put(1, 2);
23: map.put(2, 4);
24: Integer original = map.replace(2, 10); // this method return older value
- 4
25: System.out.println(map);    // {1=2, 2=10}
26: map.replaceAll((k, v) -> k + v);
27: System.out.println(map);    // {1=3, 2=12}
```

`putIfAbsent()`

```
// set a value in the map but skips it if the value is already set to a non-
null value

Map<String, String> favorites = new HashMap<>();
favorites.put("Jenny", "Bus Tour");
favorites.put("Tom", null);
favorites.putIfAbsent("Jenny", "Tram");
favorites.putIfAbsent("Sam", "Tram");
favorites.putIfAbsent("Tom", "Tram");
System.out.println(favorites); // {Tom=Tram, Jenny=Bus Tour, Sam=Tram}
```

Merging data

```
11: BiFunction<String, String, String> mapper = (v1, v2)
12:    -> v1.length()> v2.length() ? v1: v2;
13:
14: Map<String, String> favorites = new HashMap<>();
15: favorites.put("Jenny", "Bus Tour");
16: favorites.put("Tom", "Tram");
17:
18: String jenny = favorites.merge("Jenny", "Skyride", mapper);
19: String tom = favorites.merge("Tom", "Skyride", mapper);
20:
21: System.out.println(favorites); // {Tom=Skyride, Jenny=Bus Tour}
22: System.out.println(jenny);     // Bus Tour
23: System.out.println(tom);       // Skyride
```

The code on lines 11 and 12 takes two parameters and returns a value. Our implementation returns the one with the longest name. Line 18 calls this mapping function, and it sees that `Bus Tour` is longer than `Skyride`, so it leaves the value as `Bus Tour`. Line 19 calls this mapping function again. This time, `Tram` is shorter than `Skyride`, so the map is updated. Line 21 prints out the new map contents. Lines 22 and 23 show that the result is returned from `merge()`.

TABLE 9.7 Behavior of the `merge()` method

If the requested key	And mapping function returns	Then:
Has a `null` value in map	N/A (mapping function not called)	Update key's value in map with value parameter.
Has a non-`null` value in map	`null`	Remove key from map.
Has a non-`null` value in map	A non-`null` value	Set value to mapping function result.
Is not in map	N/A (mapping function not called)	Add key with value parameter to map directly without calling mapping function.

Sorting data

Most of the time, we'll use `Collections.sort()`, which return `void` because the method parameter is what gets sorted.

`Comparable` class

```
public interface Comparable<T> {
    int compareTo(T o);
```

```
}

----------example of impementation--------------
public record Duck(String name) implements Comparable<Duck> {
   public int compareTo(Duck d) {
      return name.compareTo(d.name);  // Sorts ascendingly by name
   }
}
-----
11: var ducks = new ArrayList<Duck>();
12: ducks.add(new Duck("Quack"));
13: ducks.add(new Duck("Puddles"));
14: Collections.sort(ducks);  // sort by name
15: System.out.print(ducks);  // [Duck[name=Puddles], Duck[name=Quack]]
```

Designing a compareTo() method

The most important part is the return value.
Rules:

- The number `0` is returned when the current object is equivalent to the argument to `compareTo()`.
- A negative number (less than `0`) is returned when the current object is smaller than the argument to `compareTo()`.
- A positive number (greater than `0`) is returned when the current object is larger than the argument to `compareTo()`.

```
2: public record ZooDuck(int id, String name) implements Comparable<ZooDuck>
{
3:    public int compareTo(ZooDuck d) {
4:       return id - d.id;  // Sorts ascendingly by id
5:    }
6: }

-----
21: var d1 = new ZooDuck (5, "Daffy");
22: var d2 = new ZooDuck(7, "Donald");
23: System.out.println(d1.compareTo(d2));   // -2
24: System.out.println(d1.compareTo(d1));   // 0
25: System.out.println(d2.compareTo(d1));   // 2
```

Casting the *compareTo()* Argument

When dealing with legacy code or code that does not use generics, the `compareTo()` method requires a cast since it is passed an `Object`.

```
public record LegacyDuck(String name) implements Comparable {
   public int compareTo(Object obj) {
      if(obj instanceof LegacyDuck d)
         return name.compareTo(d.name);
      throw new UnsupportedOperationException("Not a duck");
   } }
```

Checking for `null`

```
public record MissingDuck(String name) implements Comparable<MissingDuck> {
   public int compareTo(MissingDuck quack) {
      if (quack == null)
         throw new IllegalArgumentException("Poorly formed duck!");
      if (this.name == null && quack.name == null)
         return 0;
      else if (this.name == null) return -1;
      else if (quack.name == null) return 1;
      else return name.compareTo(quack.name);
   } }
```

Keeping `compareTo()` and `equals()` consistent

If you write a class that implements `Comparable`, you introduce new business logic for determining equality. The `compareTo()` method returns `0` if two objects are equal, while your `equals()` method returns `true` if two objects are equal. A *natural ordering* that uses `compareTo()` is said to be *consistent with equals* if, and only if, `x.equals(y)` is `true` whenever `x.compareTo(y)` equals `0`.

```
public class Product implements Comparable<Product> {
   private int id;
   private String name;
   public int hashCode() { return id; }

   public boolean equals(Object obj) {
      if (obj instanceof Product other)
         return this.id == other.id;
      return false;
   }

   public int compareTo(Product obj) {
      return this.name.compareTo(obj.name);
   } }
```

Comparing Data with a *Comparator*

`Comparator` is a functional interface since there is only one abstract method to implement.

```
11: Comparator<Duck> byWeight = new Comparator<>() {
12:    public int compare(Duck d1, Duck d2) {
13:       return d1.weight() - d2.weight();
14:    }
15: };
16: var ducks = new ArrayList<Duck>();
17: ducks.add(new Duck("Quack", 7));
18: ducks.add(new Duck("Puddles", 10));
19: Collections.sort(ducks);
20: System.out.println(ducks);  // [Puddles, Quack]
21: Collections.sort(ducks, byWeight);
22: System.out.println(ducks);  // [Quack, Puddles]
```

```
Comparator<Duck> byWeight = (d1, d2) -> d1.weight()-d2.weight();
Comparator<Duck> byWeight = Comparator.comparing(Duck::weight);
```

`Comparator.comparing()` is a `static` interface method that creates a `Comparator` given a lambda expression or method reference.

Comparing *Comparable* and *Comparator*

Difference	Comparable	Comparator
Package name	java.lang	java.util
Interface must be implemented by class comparing?	Yes	No
Method name in interface	compareTo()	compare()
Number of parameters	1	2
Common to declare using a lambda	No	Yes

Comparing Multiple Fields

```
public record Squirrel(int weight, String species) {}

public class MultiFieldComparator implements Comparator<Squirrel> {
   public int compare(Squirrel s1, Squirrel s2) {
      int result = s1.species().compareTo(s2.species());
      if (result != 0) return result;
      else return s1.weight() - s2.weight();
```

```
    } }
    //alternativ
    Comparator<Squirrel> c =
Comparator.comparing(Squirrel::species).thenComparingInt(Squirrel::weight);
```

TABLE 9.9 Helper static methods for building a `Comparator`

Method	Description
`comparing(function)`	Compare by results of function that returns any `Object` (or primitive autoboxed into `Object`) .
`comparingDouble(function)`	Compare by results of function that returns `double`.
`comparingInt(function)`	Compare by results of function that returns `int`.
`comparingLong(function)`	Compare by results of function that returns `long`.
`naturalOrder()`	Sort using order specified by the `Comparable` implementation on the object itself.
`reverseOrder()`	Sort using reverse of order specified by `Comparable` implementation on the object itself.

TABLE 9.10 Helper default methods for building a `Comparator`

Method	Description
`reversed()`	Reverse order of chained `Comparator`.
`thenComparing(function)`	If previous `Comparator` returns `0`, use this comparator function that returns `Object` or can be autoboxed into one. Otherwise, return result from previous Comparator.
`thenComparingDouble(function)`	If previous `Comparator` returns `0`, use this comparator function that returns `double`. Otherwise, return result from previous `Comparator`.
`thenComparingInt(function)`	If previous `Comparator` returns `0`, use this comparator function that returns `int`. Otherwise, return result from previous `Comparator`.
`thenComparingLong(function)`	If previous `Comparator` returns `0`, use this comparator function that returns `long`. Otherwise, return result from previous `Comparator`.

Sorting and Searching

Now that you've learned all about `Comparable` and `Comparator`, we can finally do something useful with them, like sorting. The `Collections.sort()` method uses the `compareTo()` method to sort. It expects the objects to be sorted to be `Comparable`.
If your object doesn't implement the Comparable, you can fix this by passing a `Comparator` to `sort()`. Remember that a `Comparator` is useful when you want to specify sort order without using a `compareTo()` method.

Sequenced collections

Sequneced collection

Is a collection in which the encounter order is well-defined.
By *encounder order*, it means all of the elements can be read in a repeatable way.

SequencedCollection

Method	Description
`addFirst(E e)`	Adds element as the first element in the collection
`addLast(E e)`	Adds element as the last element in the collection
`getFirst()`	Retrieves the first element in the collection
`getLast()`	Retrieves the last element in the collection
`removeFirst()`	Removes the first element in the collection
`removeLast()`	Removes the last element in the collection
`reversed()`	Returns a reverse-ordered view of the collection

A `SequencedSet` is a subtype of `SequencedCollection`; therefore, it inherits all its methods. It only applies to `SequencedCollection` classes that also implement `Set`, such as `LinkedHashSet` and `TreeSet`.

SequencedMap

SequencedMap

Is a subtype of `SequencedCollection`; therefore, it inherits all its methods. It only applies to `SequencedCollection` classes that also implement `Set`, such as `LinkedHashSet` and `TreeSet`.

Method	Description
`firstEntry()`	Retrieves the first key/value pair in the map
`lastEntry()`	Retrieves the last key/value pair in the map
`pollFirstEntry()`	Removes and retrieves the first key/value pair in the map
`pollLastEntry()`	Removes and retrieves the last key/value pair in the map
`putFirst(K k, V v)`	Adds the key/value pair as the first element in the map
`putLast(K k, V v)`	Adds the key/value pair as the last element in the map
`reversed()`	Returns a reverse-ordered view of the map

Reviewing Collection Types

> An *unmodifiable view* is a wrapper object around a collection that cannot be modified through the view itself. While the view object cannot be modified, the underlying data can still be modified.

```
10: Map<String, Integer> map = new TreeMap<>();
11: map.put("blue", 41);
12: map.put("red", 90);
13: List<String> list = Arrays.asList("green", "yellow");
14: Set<String> set = new HashSet<>(list);
15:
16: Map<String, Integer> mapView = Collections.unmodifiableMap(map);
18: Collection<String> collView  = Collections.unmodifiableCollection(list);
19: List<String> listView        = Collections.unmodifiableList(list);
20: Set<String> setView          = Collections.unmodifiableSet(set);

--each line from below throw an error
collView.add("pink");
setView.remove("green");
mapView.put("blue", 42);
```

However, since it is a view, nothing prevents you from changing the original values. For example:

```
24: System.out.println(mapView);   // {blue=41, red=90}
25: System.out.println(collView);  // [green, yellow]
26: System.out.println(listView);  // [green, yellow]
27: System.out.println(setView);   // [green, yellow]
28:
29: map.put("blue", 105);
30: list.set(1, "purple");
31:
32: System.out.println(mapView);   // {blue=105, red=90}
33: System.out.println(collView);  // [green, purple]
34: System.out.println(listView);  // [green, purple]
35: System.out.println(setView);   // [green, yellow]
```

However, `setView` has not changed value. The constructor on line 14 makes a new set that is disconnected from the original data structure. This means line 30 has no effect on `set`.

Comparing Collection Types

Type	Can contain duplicate elements?	Elements always ordered?	Has keys and values?	Must add/remove in specific order?
List	Yes	Yes (by index)	No	No
Queue	Yes	Yes (retrieved in defined order)	No	Yes
Set	No	No	No	No
Map	Yes (for values)	No	Yes	No

Type	Java Collections Framework interfaces	Ordered?	Sorted?	Calls hashCode?	Calls compareTo?
ArrayDeque	Deque SequencedCollection	Yes	No	No	No
ArrayList	List SequencedCollection	Yes	No	No	No
HashMap	Map	No	No	Yes	No
HashSet	Set	No	No	Yes	No
LinkedList	Deque List SequencedCollection	Yes	No	No	No

LinkedHashSet	Set SequencedSet	Yes	No	No	No
LinkedHashMap	Map SequencedMap	Yes	No	No	No
TreeMap	Map SequencedMap	Yes	Yes	No	Yes
TreeSet	Set SequencedCollection SequencedSet	Yes	Yes	No	Yes

Generics

Creating Generic classes

```
public class Crate<T> {
   private T contents;
   public T lookInCrate() {
      return contents;
   }
   public void packCrate(T contents) {
      this.contents = contents;
   }
}
```

A type parameter can be named anything you want. The convention is to use single uppercase letters to make it obvious that they aren't real class names. The following are common letters to use:

- `E` for an element
- `K` for a map key
- `V` for a map value
- `N` for a number
- `T` for a generic data type
- `S`, `U`, `V`, and so forth for multiple generic types

Generic classes aren't limited to having a single type parameter. This class shows two generic parameters.

```
public class SizeLimitedCrate<T, U> {
   private T contents;
   private U sizeLimit;
```

```
   public SizeLimitedCrate(T contents, U sizeLimit) {
      this.contents = contents;
      this.sizeLimit = sizeLimit;
   } }
```

Overloading a Generic Method

```
public class Anteater extends LongTailAnimal {
   protected void chew(List<Object> input) {}
   protected void chew(ArrayList<Double> input) {}
}
```

The first `chew()` method compiles because it uses the same generic type in the overridden method as the one defined in the parent class. The second `chew()` method compiles as well. However, it is an overloaded method because one of the method arguments is a `List` and the other is an `ArrayList`. When working with generic methods, it's important to consider the underlying type.

Returning Generic Types

When you're working with overridden methods that return generics, the return values must be covariant. In terms of generics, this means the return type of the class or interface declared in the overriding method must be a subtype of the class defined in the parent class. The generic parameter type must match its parent's type exactly.

Implementing Generic Interfaces

```
public interface Shippable<T> {
   void ship(T t);
}

class ShippableRobotCrate implements Shippable<Robot> {
   public void ship(Robot t) { }
}

class ShippableAbstractCrate<U> implements Shippable<U> {
   public void ship(U t) { }
}
```

What u can't do with generic

Most of the limitations are due to type erasure. Oracle refers to a type whose information is fully available at runtime as a *reifiable type*. Reifiable types can do anything that Java allows. Non-reifiable types have some limitations.

Here are the things that you can't do with generics (and by "can't," we mean without resorting to contortions like passing in a class object):

- **Call a constructor:** Writing `new T()` is not allowed because at runtime, it would be `new Object()`.
- **Create an array of that generic type:** This one is the most annoying, but it makes sense because you'd be creating an array of `Object` values.
- **Call `instanceof`:** This is not allowed because at runtime `List<Integer>` and `List<String>` look the same to Java, thanks to type erasure.
- **Use a primitive type as a generic type parameter:** This isn't a big deal because you can use the wrapper class instead. If you want a type of `int`, just use `Integer`.
- **Create a `static` variable as a generic type parameter:** This is not allowed because the type is linked to the instance of the class.
- **Catch an exception of type T**: Even if `T` extends `Exception`, it cannot be used in a catch block since the precise type is not known.

Writing Generic methods

This is often useful for `static` methods since they aren't part of an instance that can declare the type. However, it is also allowed on non-`static` methods.

```
public class Handler {
   public static <T> void prepare(T t) {
      System.out.println("Preparing " + t);
   }

   public static <T> Crate<T> ship(T t) {
      System.out.println("Shipping " + t);
      return new Crate<T>();
   }
}
```

```
2: public class More {
3:    public static <T> void sink(T t) { }
4:    public static <T> T identity(T t) { return t; }
5:    public static T noGood(T t) { return t; } // DOES NOT COMPILE
6: }
```

Creating a Generic Record

```
public record CrateRecord<T>(T contents) {
   @Override
   public T contents() {
      if (contents == null)
         throw new IllegalStateException("missing contents");
```

```
      return contents;
   }
}
```

Bounding Generic Types

> A *bounded parameter type* is a generic type that specifies a bound for the generic. Be warned that this is the hardest section in the chapter, so don't feel bad if you have to read it more than once.

A *wildcard generic type* is an unknown generic type represented with a question mark (`?`). You can use generic wildcards in three ways. This section looks at each of these three wildcard types.

Type of bound	Syntax	Example
Unbounded wildcard	`?`	`List<?> a = new ArrayList<String>();`
Wildcard with upper bound	`? extends type`	`List<? extends Exception> a = new ArrayList<RuntimeException>();`
Wildcard with lower bound	`? super type`	`List<? super Exception> a = new ArrayList<Object>();`

Creating Unbounded Wildcards

```
public static void printList(List<?> list) {
   for (Object x: list)
      System.out.println(x);
}

public static void main(String[] args) {
   List<String> keywords = new ArrayList<>();
   keywords.add("java");
   printList(keywords);
}
```

Creating Upper-Bounded Wildcards

```
ArrayList<Number> list = new ArrayList<Integer>(); // DOES NOT COMPILE

List<? extends Number> list = new ArrayList<Integer>();

// The upper-bounded wildcard says that any class that `extends Number` or
`Number` itself can be used as the formal parameter type:

public static long total(List<? extends Number> list) {
   long count = 0;
```

```
    for (Number number: list)
       count += number.longValue();
    return count;
}
```

Lower bound

TABLE 9.16 Why we need a lower bound

`static void addSound( list) { list.add("quack"); }`	**Method compiles**	**Can pass a** `List<String>`	**Can pass a** `List<Object>`
`List<?>`	No	Yes	Yes
`List<? extends Object>`	No	Yes	Yes
`List<Object>`	Yes	No (with generics, must pass exact match)	Yes
`List<? super String>`	Yes	Yes	Yes