Chapter 14 - Generics and Collections

Using Method References

There are four formats for method references:

- Static methods
- Instance methods on a particular instance
- Instance methods on a parameter to be determined at runtime
- Constructors

Calling Static Methods

The `Collections` class has a `static` method that can be used for sorting. Per Table 14.1, the `Consumer` functional interface takes one parameter and does not return anything. Here we will assign a method reference and a lambda to this functional interface:

```java
14: Consumer<List<Integer>> methodRef = Collections::sort;
15: Consumer<List<Integer>> lambda = x -> Collections.sort(x);
```

Calling Instance Methods on a Particular Object

The `String` class has a `startsWith()` method that takes one parameter and returns a `boolean`. Conveniently, a `Predicate` is a functional interface that takes one parameter and returns a `boolean`. Let's look at how to use method references with this code:

```java
18: var str = "abc";
19: Predicate<String> methodRef = str::startsWith;
20: Predicate<String> lambda = s -> str.startsWith(s);
```

Calling Instance Methods on a Parameter

This time, we are going to call an instance method that doesn't take any parameters. The trick is that we will do so without knowing the instance in advance.

```java
23: Predicate<String> methodRef = String::isEmpty;
24: Predicate<String> lambda = s -> s.isEmpty();
```

Line 23 says the method that we want to call is declared in `String`. It looks like a `static` method, but it isn't. Instead, Java knows that `isEmpty()` is an instance method that does not take any parameters. Java uses the parameter supplied at runtime as the instance on which the method is called.

Compare lines 23 and 24 with lines 19 and 20 of our instance example. They look similar, although one references a local variable named `str`, while the other only references the functional interface parameters.

You can even combine the two types of instance method references. We are going to use a functional interface called a `BiPredicate`, which takes two parameters and returns a `boolean`.

```
26: BiPredicate<String, String> methodRef = String::startsWith;
27: BiPredicate<String, String> lambda = (s, p) -> s.startsWith(p);
```

Since the functional interface takes two parameters, Java has to figure out what they represent. The first one will always be the instance of the object for instance methods. Any others are to be method parameters.

Remember that line 26 may look like a `static` method, but it is really a method reference declaring that the instance of the object will be specified later. Line 27 shows some of the power of a method reference. We were able to replace two lambda parameters this time.

Calling Constructors

A *constructor reference* is a special type of method reference that uses `new` instead of a method, and it instantiates an object. It is common for a constructor reference to use a `Supplier` as shown here:

```
30: Supplier<List<String>> methodRef = ArrayList::new;
31: Supplier<List<String>> lambda = () -> new ArrayList();
```

Reviewing Method References

TABLE 14.2 Method references

Type	Before colon	After colon	Example
Static methods	Class name	Method name	`Collections::sort`
Instance methods on a particular object	Instance variable name	Method name	`str::startsWith`
Instance methods on a parameter	Class name	Method name	`String::isEmpty`
Constructor	Class name	`new`	`ArrayList::new`

Using Wrapper Classes

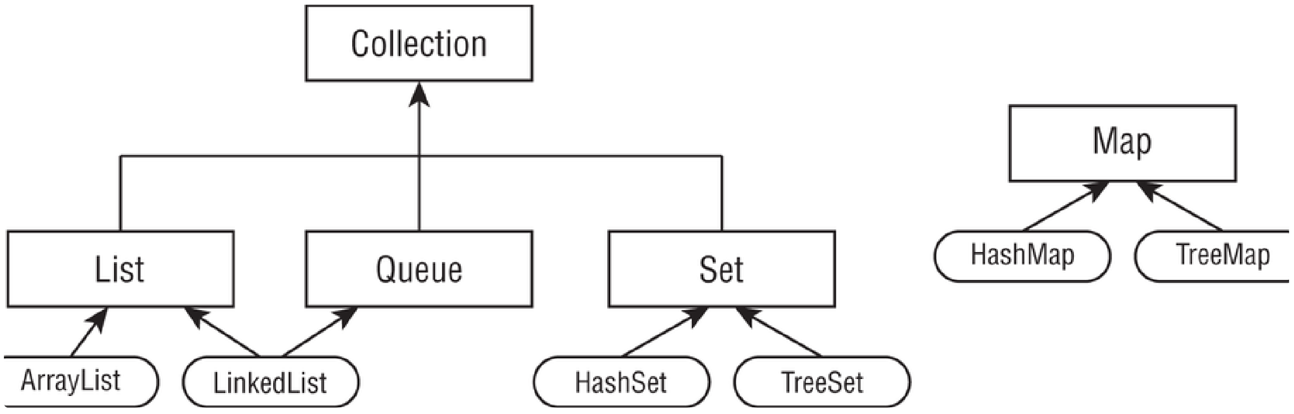
Primitive type	Wrapper class	Example of initializing
`boolean`	`Boolean`	`Boolean.valueOf(true)`
`byte`	`Byte`	`Byte.valueOf((byte) 1)`
`short`	`Short`	`Short.valueOf((short) 1)`
`int`	`Integer`	`Integer.valueOf(1)`
`long`	`Long`	`Long.valueOf(1)`
`float`	`Float`	`Float.valueOf((float) 1.0)`
`double`	`Double`	`Double.valueOf(1.0)`
`char`	`Character`	`Character.valueOf('c')`

There are two tricks in the space of autoboxing and unboxing. The first has to do with `null` values. This innocuous-looking code throws an exception:

```
15: var heights = new ArrayList<Integer>();
16: heights.add(null);
17: int h = heights.get(0); // NullPointerException
```

second trick. The `remove()` method is overloaded. One signature takes an `int` as the index of the element to remove. The other takes an `Object` that should be removed. On line 27, Java sees a matching signature for `int`, so it doesn't need to autobox the call to the method.

Using Lists, Sets, Maps, and Queues



Common Collections Methods

add()

The `add()` method inserts a new element into the `Collection` and returns whether it was successful. The method signature is as follows:

```
boolean add(E element)
```

A `List` allows duplicates, making the return value `true` each time. A `Set` does not allow duplicates. On line 9, we tried to add a duplicate so that Java returns `false` from the `add()` method.

remove()

The `remove()` method removes a single matching value in the `Collection` and returns whether it was successful. The method signature is as follows:

```
boolean remove(Object object)
```

Java does not allow removing elements from a list while using the enhanced `for` loop.

```
Collection<String> birds = new ArrayList<>();
birds.add("hawk");
birds.add("hawk");
birds.add("hawk");

for (String bird : birds) // ConcurrentModificationException
   birds.remove(bird);
```

isEmpty() and size()

The `isEmpty()` and `size()` methods look at how many elements are in the `Collection`. The method signatures are as follows:

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

clear()

The `clear()` method provides an easy way to discard all elements of the `Collection`. The method signature is as follows:

```
void clear()
```

contains()

The `contains()` method checks whether a certain value is in the `Collection`. The method signature is as follows:

```
boolean contains(Object object)
```

removeIf()

The `removeIf()` method removes all elements that match a condition. We can specify what should be deleted using a block of code or even a method reference.

The method signature looks like the following. (We will explain what the `? super` means in the “Working with Generics” section later in this chapter.)

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

It uses a `Predicate`, which takes one parameter and returns a `boolean`. Let's take a look at an 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()

Looping through a `Collection` is common. On the 1Z0-815 exam, you wrote lots of loops. There's also a `forEach()` method that you can call on a `Collection`. It uses a `Consumer` that takes a single parameter and doesn't return anything. The method signature is as follows:

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

Using the List Interface

The main thing that all `List` implementations have in common is that they are ordered and allow duplicates.

The main benefit of an `ArrayList` is that you can look up any element in constant time. Adding or removing an element is slower than accessing an element. This makes an `ArrayList` a good choice when you are reading more often than (or the same amount as) writing to the `ArrayList`.

A `LinkedList` is special because it implements both `List` and `Queue`. It has all the methods of a `List`. It also has additional methods to facilitate adding or removing from the beginning and/or end of the list.

The main benefits of a `LinkedList` are that you can access, add, and remove from the beginning and end of the list in constant time. The trade-off is that dealing with an arbitrary index takes linear time. This makes a `LinkedList` a good choice when you'll be using it as `Queue`. As you saw in Figure 14.1, a `LinkedList` implements both the `List` and `Queue` interface.

Factory methods to create a List

Method	Description	Can add elements?	Can replace element?	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

TABLE 14.5 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
`E remove(int index)`	Removes element at index and moves the rest toward the front
`void replaceAll(UnaryOperator<E> op)`	Replaces each element in the list with the result of the operator
`E set(int index, E e)`	Replaces element at index and returns original. Throws `IndexOutOfBoundsException` if the index is larger than the maximum one set

*Now, let's look at using the `replaceAll()` method. It takes a `UnaryOperator` that takes one parameter and returns a value of the same type.

```java
List<Integer> numbers = Arrays.asList(1, 2, 3);
numbers.replaceAll(x -> x*2);
System.out.println(numbers);   // [2, 4, 6]
```

Using the Set Interface

The main benefit is that adding elements and checking whether an element is in the set both have constant time. The trade-off is that you lose the order in which you inserted the elements. Most of the time, you aren't concerned with this in a set anyway, making `HashSet` the most common set.

A `TreeSet` stores its elements in a sorted tree structure. The main benefit is that the set is always in sorted order. The trade-off is that adding and checking whether an element exists take longer than with a `HashSet`, especially as the tree grows larger.

Remember that the `equals()` method is used to determine equality. The `hashCode()` method is used to know which bucket to look in so that Java doesn't have to look through the whole set to find out whether an object is there. The best case is that hash codes are unique, and Java has to call `equals()` on only one object. The worst case is that all implementations return the same `hashCode()`, and Java has to call `equals()` on every element of the set anyway.

Using the Queue Interface

You use a queue when elements are added and removed in a specific order. Queues are typically used for sorting elements prior to processing them.

Comparing *Queue* Implementations

You saw `LinkedList` earlier in the `List` section. In addition to being a list, it is a double-ended queue.

TABLE 14.6 Queue methods

Method	Description	Throws exception on failure
`boolean add(E e)`	Adds an element to the back of the queue and returns `true` or throws an exception	Yes
`E element()`	Returns next element or throws an exception if empty queue	Yes
`boolean offer(E e)`	Adds an element to the back of the queue and returns whether successful	No
`E remove()`	Removes and returns next element or throws an exception if empty queue	Yes
`E poll()`	Removes and returns next element or returns `null` if empty queue	No
`E peek()`	Returns next element or returns `null` if empty queue	No

As you can see, there are basically two sets of methods. One set throws an exception when something goes wrong. The other uses a different return value when something goes wrong. The `offer()` / `poll()` / `peek()` methods are more common. This is the standard language people use when working with queues.

Comparing *Map* Implementations

A `HashMap` stores the keys in a hash table. This means that it uses the `hashCode()` method of the keys to retrieve their values more efficiently.

The main benefit is that adding elements and retrieving the element by key both have constant time. The trade-off is that you lose the order in which you inserted the elements. Most of the time, you aren't concerned with this in a map anyway. If you were, you could use `LinkedHashMap`, but that's not in scope for the exam.

A `TreeMap` stores the keys in a sorted tree structure. The main benefit is that the keys are always in sorted order. Like a `TreeSet`, the trade-off is that adding and checking whether a key is present takes longer as the tree grows larger.

TABLE 14.7 Map methods

Method	Description
`void clear()`	Removes all keys and values from the 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 a `Set` of key/value pairs.
`void forEach(BiConsumer(K key, V value))`	Loop through each key/value pair.
`V get(Object key)`	Returns the value mapped by key or `null` if none is mapped.
`V getOrDefault(Object key, V defaultValue)`	Returns the value mapped by the key or the default value if none is mapped.
`boolean isEmpty()`	Returns whether the map is empty.
`Set<K> keySet()`	Returns set of all keys.
`V merge(K key, V value, Function(<V, V, V> func))`	Sets value if key not set. Runs the function if the key is set to determine the new value. Removes if `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 the value for a given key if the key is set. Returns the original value or `null` if none.
`void replaceAll(BiFunction<K, V, V> func)`	Replaces each value with the results of the function.
`int size()`	Returns the number of entries (key/value pairs) in the map.
`Collection<V> values()`	Returns `Collection` of all values.

The `contains()` method is on the `Collection` interface but not the `Map` interface

forEach() and entrySet()

Interestingly, if you don't care about the key, this particular code could have been written with the `values()` method and a method reference instead.

```
map.values().forEach(System.out::println);
```

Another way of going through all the data in a map is to get the key/value pairs in a `Set`. Java has a `static` interface inside `Map` called `Entry`. It provides methods to get the key and value of each pair.

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

getOrDefault()

The `get()` method returns null if the requested key is not in map. Sometimes you prefer to have a different value returned. Luckily, the `getOrDefault()` method makes this easy. Let's compare the two methods.

```
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', ""));
7: System.out.println("Y marks the " + map.get('y'));
8: System.out.println("Y marks the " + map.getOrDefault('y', ""));
```

This code prints the following:

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

replace() and replaceAll()

These methods are similar to the `Collection` version except a key is involved.

```
21: Map<Integer, Integer> map = new HashMap<>();
22: map.put(1, 2);
23: map.put(2, 4);
24: Integer original = map.replace(2, 10); // 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()

The `putIfAbsent()` method sets 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}
```

merge()

The `merge()` method adds logic of what to choose. Suppose we want to choose the ride with the longest name. We can write code to express this by passing a mapping function to the `merge()` method.

```
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 `merge()` method also has logic for what happens if `null` values or missing keys are involved. In this case, it doesn't call the `BiFunction` at all, and it simply uses the new value.

```
BiFunction<String, String, String> mapper =
   (v1, v2) -> v1.length()> v2.length() ? v1 : v2;
Map<String, String> favorites = new HashMap<>();
favorites.put("Sam", null);
favorites.merge("Tom", "Skyride", mapper);
favorites.merge("Sam", "Skyride", mapper);
System.out.println(favorites);   // {Tom=Skyride, Sam=Skyride}
```

Notice that the mapping function isn't called. If it were, we'd have a `NullPointerException`. The mapping function is used only when there are two actual values to decide between.

The final thing to know about `merge()` is what happens when the mapping function is called and returns `null`. The key is removed from the map when this happens:

```
BiFunction<String, String, String> mapper = (v1, v2) -> null;
Map<String, String> favorites = new HashMap<>();
favorites.put("Jenny", "Bus Tour");
favorites.put("Tom", "Bus Tour");

favorites.merge("Jenny", "Skyride", mapper);
favorites.merge("Sam", "Skyride", mapper);
System.out.println(favorites);   // {Tom=Bus Tour, Sam=Skyride}
```

TABLE 14.8 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 key 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.

Creating a *Comparable* Class

The `Comparable` interface has only one method. In fact, this is the entire interface:

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

We still need to know what the `compareTo()` method returns so that we can write our own. There are three rules to know.

- 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()`.

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`.

Similarly, `x.equals(y)` must be `false` whenever `x.compareTo(y)` is not `0`. You are strongly encouraged to make your `Comparable` classes consistent with equals because not all collection classes behave predictably if the `compareTo()` and `equals()` methods are not consistent.

Comparing Data with a *Comparator*

Sometimes you want to sort an object that did not implement `Comparable`, or you want to sort objects in different ways at different times. Suppose that we add `weight` to our `Duck` class. We now have the following:

```
16:
17:    public static void main(String[] args) {
18:       Comparator<Duck> byWeight = new Comparator<Duck>() {
19:          public int compare(Duck d1, Duck d2) {
20:             return d1.getWeight()-d2.getWeight();
21:          }
22:       };
```

`Comparable` and `Comparator` are in different packages, namely, `java.lang` versus `java.util`, respectively.

`Comparator` is a functional interface since there is only one abstract method to implement. This means that we can rewrite the comparator on lines 18-22 using a lambda expression, as shown here:

```
Comparator<Duck> byWeight = (d1, d2) -> d1.getWeight()-d2.getWeight();
```

Alternatively, we can use a method reference and a helper method to specify we want to sort by weight.

```
Comparator<Duck> byWeight = Comparator.comparing(Duck::getWeight);
```

TABLE 14.11 Comparison of 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

Alternatively, we can use method references and build the comparator. This code represents logic for the same comparison.

```
Comparator<Squirrel> c = Comparator.comparing(Squirrel::getSpecies)
   .thenComparingInt(Squirrel::getWeight);
```

Suppose we want to sort in descending order by species.

```
var c = Comparator.comparing(Squirrel::getSpecies).reversed();
```

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

Sorting and Searching

Now that you've learned all about `Comparable` and `Comparator`, we can finally do something useful with it, like sorting. The `Collections.sort()` method uses the `compareTo()` method to sort. It expects the objects to be sorted to be `Comparable`.

```
2: public class SortRabbits {
3:    static class Rabbit{ int id; }
4:    public static void main(String[] args) {
5:       List<Rabbit> rabbits = new ArrayList<>();
6:       rabbits.add(new Rabbit());
7:       Collections.sort(rabbits); // DOES NOT COMPILE
8:    } }
```

There is a trick in working with `binarySearch()`. What do you think the following outputs?

```
3: var names = Arrays.asList("Fluffy", "Hoppy");
4: Comparator<String> c = Comparator.reverseOrder();
5: var index = Collections.binarySearch(names, "Hoppy", c);
6: System.out.println(index);
```

The correct answer is `-1`. Before you panic, you don't need to know that the answer is `-1`. You do need to know that the answer is not defined. Line 3 creates a list, `[Fluffy, Hoppy]`. This list happens to be sorted in ascending order. Line 4 creates a `Comparator` that reverses the natural order. Line 5 requests a binary search in descending order. Since the list is in ascending order, we don't meet the precondition for doing a search.

Going back to our `Rabbit` that does not implement `Comparable`, we try to add it to a `TreeSet`.

```
2:  public class UseTreeSet {
3:     static class Rabbit{ int id; }
4:     public static void main(String[] args) {
5:        Set<Duck> ducks = new TreeSet<>();
6:        ducks.add(new Duck("Puddles"));
7:
8:        Set<Rabbit> rabbits = new TreeSet<>();
9:        rabbits.add(new Rabbit());  // ClassCastException
10: } }
```

Line 6 is fine. `Duck` does implement `Comparable`. `TreeSet` is able to sort it into the proper position in the set. Line 9 is a problem. When `TreeSet` tries to sort it, Java discovers the fact that `Rabbit` does not implement `Comparable`. Java throws an exception that looks like this:

```
Exception in thread "main" java.lang.ClassCastException:
   class Duck cannot be cast to class java.lang.Comparable
```

Generic Interfaces

Just like a class, an interface can declare a formal type parameter. For example, the following `Shippable` interface uses a generic type as the argument to its `ship()` method:

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

There are three ways a class can approach implementing this interface. The first is to specify the generic type in the class. The following concrete class says that it deals only with robots. This lets it declare the `ship()` method with a `Robot` parameter.

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

The next way is to create a generic class. The following concrete class allows the caller to specify the type of the generic:

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

Raw Types

The final way is to not use generics at all. This is the old way of writing code. It generates a compiler warning about `Shippable` being a *raw type*, but it does compile. Here the `ship()` method has an `Object` parameter since the generic type is not defined:

```
class ShippableCrate implements Shippable {
   public void ship(Object t) { }
}
```

Generic Methods

```
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: }
```

Line 3 shows the formal parameter type immediately before the return type of `void`. Line 4 shows the return type being the formal parameter type. It looks weird, but it is correct. Line 5 omits the formal parameter type, and therefore it does not compile.

When you have a method declare a generic parameter type, it is independent of the class generics. Take a look at this class that declares a generic `T` at both levels:

```
1: public class Crate<T> {
2:    public <T> T tricky(T t) {
3:       return t;
4:    }
5: }
```

See if you can figure out the type of `T` on lines 1 and 2 when we call the code as follows:

```
10: public static String createName() {
11:    Crate<Robot> crate = new Crate<>();
12:    return crate.tricky("bot");
13: }
```

Clearly, “T is for tricky.” Let's see what is happening. On line 1, `T` is `Robot` because that is what gets referenced when constructing a `Crate`. On line 2, `T` is `String` because that is what is passed to the method. When you see code like this, take a deep breath and write down what is happening so you don't get confused.

TABLE 14.14 Types of bounds

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

Unbounded Wildcards

An unbounded wildcard represents any data type. You use `?` when you want to specify that any type is okay with you. Let's suppose that we want to write a method that looks through a list of any type.

```
public static void printList(List<Object> 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); // DOES NOT COMPILE
}
```

Wait. What's wrong? A `String` is a subclass of an `Object`. This is true. However, `List<String>` cannot be assigned to `List<Object>`. We know, it doesn't sound logical. Java is trying to protect us from ourselves with this one. Imagine if we could write code like this:

```
4: List<Integer> numbers = new ArrayList<>();
5: numbers.add(new Integer(42));
6: List<Object> objects = numbers; // DOES NOT COMPILE
7: objects.add("forty two");
8: System.out.println(numbers.get(1));
```

Upper-Bounded Wildcards

Let's try to write a method that adds up the total of a list of numbers. We've established that a generic type can't just use a subclass.

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

Instead, we need to use a wildcard.

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

Something interesting happens when we work with upper bounds or unbounded wildcards. The list becomes logically immutable and therefore cannot be modified. Technically, you can remove elements from the list, but the exam won't ask about this.

```
2: static class Sparrow extends Bird { }
3: static class Bird { }
4:
5: public static void main(String[] args) {
6:    List<? extends Bird> birds = new ArrayList<Bird>();
7:    birds.add(new Sparrow()); // DOES NOT COMPILE
8:    birds.add(new Bird());    // DOES NOT COMPILE
9: }
```

The problem stems from the fact that Java doesn't know what type `List<? extends Bird>` really is. It could be `List<Bird>` or `List<Sparrow>` or some other generic type that hasn't even been written yet. Line 7 doesn't compile because we can't add a `Sparrow` to `List<? extends Bird>`, and line 8 doesn't compile because we can't add a `Bird` to `List<Sparrow>`. From Java's point of view, both scenarios are equally possible, so neither is allowed.

Note that we used the keyword `extends` rather than `implements`. Upper bounds are like anonymous classes in that they use `extends` regardless of whether we are working with a class or an interface.

TABLE 14.15 Why we need a lower bound

`public static void addSound(______list) {list.add("quack");}`	Method compiles	Can pass a `List<String>`	Can pass a `List<Object>`
`List<?>`	No (unbounded generics are immutable)	Yes	Yes
`List<? extends Object>`	No (upper-bounded generics are immutable)	Yes	Yes
`List<Object>`	Yes	No (with generics, must pass exact match)	Yes
`List<? super String>`	Yes	Yes	Yes