**Chapter 6**

# Generics and Collections

Business Training

1

## Content

- **Override hashCode(), equals(), and toString()**
- **Collections Overview**
- **Use the Collections Framework**
- **Generic Types**

*M. Romdhani, 2020*

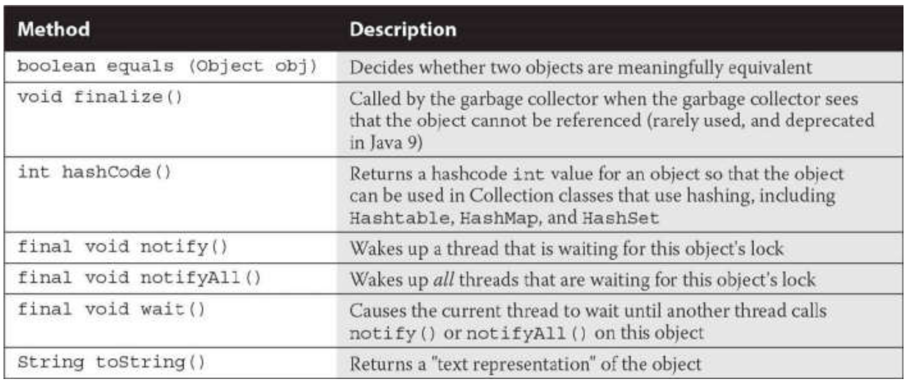
**2**

2

## Override hashCode(), equals(), and toString()

3

## Methods of Class Object Covered on the Exam



| Method | Description |
|---|---|
| `boolean equals (Object obj)` | Decides whether two objects are meaningfully equivalent |
| `void finalize()` | Called by the garbage collector when the garbage collector sees that the object cannot be referenced (rarely used, and deprecated in Java 9) |
| `int hashCode()` | Returns a hashcode `int` value for an object so that the object can be used in Collection classes that use hashing, including `Hashtable`, `HashMap`, and `HashSet` |
| `final void notify()` | Wakes up a thread that is waiting for this object's lock |
| `final void notifyAll()` | Wakes up *all* threads that are waiting for this object's lock |
| `final void wait()` | Causes the current thread to wait until another thread calls `notify()` or `notifyAll()` on this object |
| `String toString()` | Returns a "text representation" of the object |

- **Chapter 10 covers wait(), notify(), and notifyAll().**

4

## The toString() Method

- **Override toString() when you want a mere mortal to be able to read something meaningful about the objects of your class.**
  - Code can call toString() on your object when it wants to read useful details about your object.
    - When you pass an object reference to the System.out.println() method, for example, the object's toString() method is called, and the return of toString() is shown in the following example:
- **Running the HardToRead class gives us the lovely and meaningful**

```
   public class HardToRead {
     public static void main (String [] args) {
       HardToRead h = new HardToRead();
       System.out.println(h);
     }
   }
```

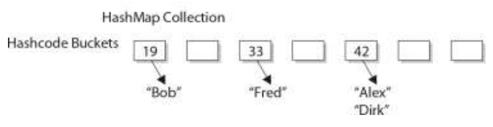
```
% java HardToRead
HardToRead@a47e0
```

  - Une implémentation de toString() est en général simple à générer via un IDE (Eclipse, Intellij IDEA, …)

## Overriding hashCode() and equals()

- **Understanding Hashcodes**
  - In order to understand what's appropriate and correct, we have to look at how some of the collections use hashcodes.
  - A simplified hashcode example

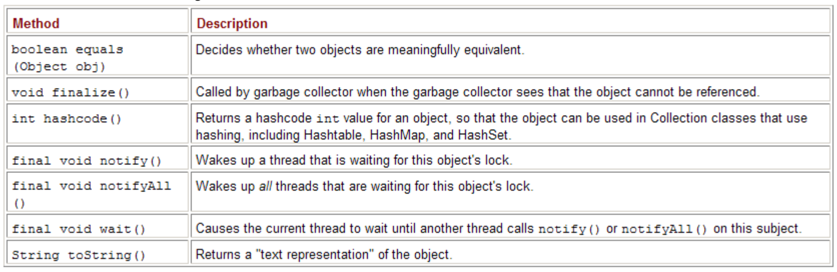
| Key | Hashcode Algorithm | Hashcode |
|---|---|---|
| Alex | A(1) + L(12) + E(5) + X(24) | = 42 |
| Bob | B(2) + O(15) + B(2) | = 19 |
| Dirk | D(4) + I(9) + R(18) + K(11) | = 42 |
| Fred | F(6) + R(18) + E(5) + D(4) | = 33 |



  - **If two objects are equal, their hashcodes must be equal as well.**

## Overriding hashCode() and equals()

- **Methods of Class Object Covered on the Exam**



| Method | Description |
|---|---|
| `boolean equals (Object obj)` | Decides whether two objects are meaningfully equivalent. |
| `void finalize()` | Called by garbage collector when the garbage collector sees that the object cannot be referenced. |
| `int hashcode()` | Returns a hashcode `int` value for an object, so that the object can be used in Collection classes that use hashing, including Hashtable, HashMap, and HashSet. |
| `final void notify()` | Wakes up a thread that is waiting for this object's lock. |
| `final void notifyAll()` | Wakes up *all* threads that are waiting for this object's lock. |
| `final void wait()` | Causes the current thread to wait until another thread calls `notify()` or `notifyAll()` on this subject. |
| `String toString()` | Returns a "text representation" of the object. |

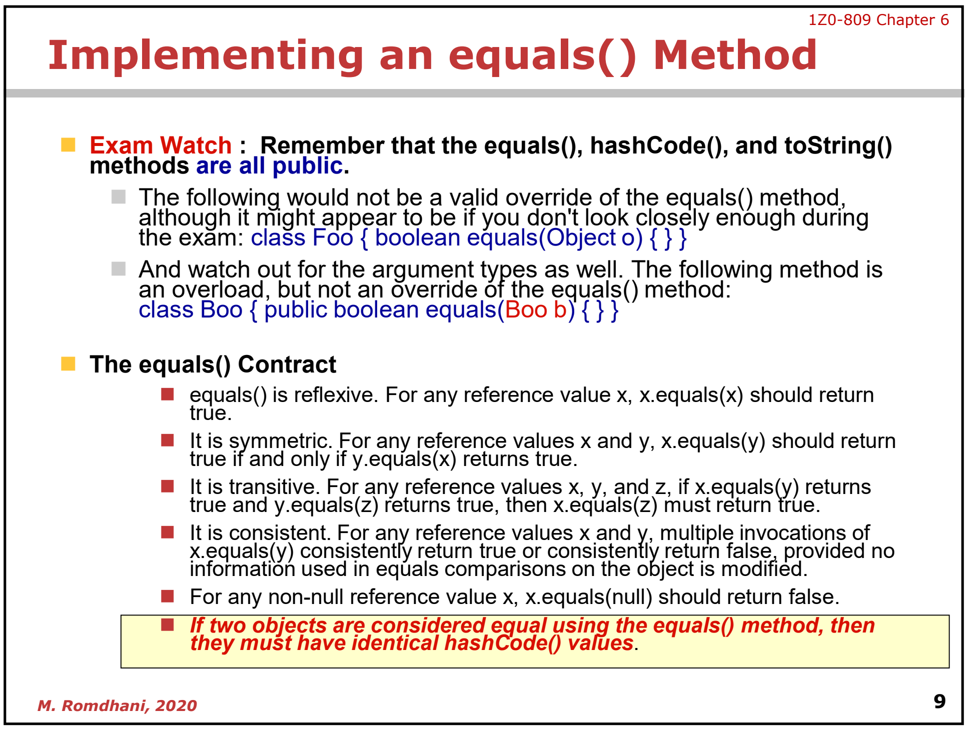
  - The toString() Method

- **Overriding equals()**
  - We discussed how comparing two object references using the == operator evaluates to true only when both references refer to the same object
  - What It Means If You don't Override equals() ?
    - If you don't override a class's equals() method, you won't be able to use those objects as a key in a hashtable and you probably won't get accurate Sets
    - The equals() method in class Object uses only the == operator for comparisons, so unless you override equals(), two objects are considered equal only if the two references refer to the same object.

## Implementing an equals() Method

- **Let's say you decide to override equals() in your class. It might look like this:**

```
public class EqualsTest {
  public static void main (String [] args) {
     Moof one = new Moof(8);
     Moof two = new Moof(8);
     if (one.equals(two)) {
        System.out.println("one and two are equal");
     }
   }
}
class Moof {
  private int moofValue;
  Moof(int val) {
     moofValue = val;
  }
  public int getMoofValue() {
     return moofValue;
  }
  public boolean equals(Object o) {
    if ((o instanceof Moof) && (((Moof)o).getMoofValue() == this.moofValue))
      return true;
     else  return false;
  }
}
```



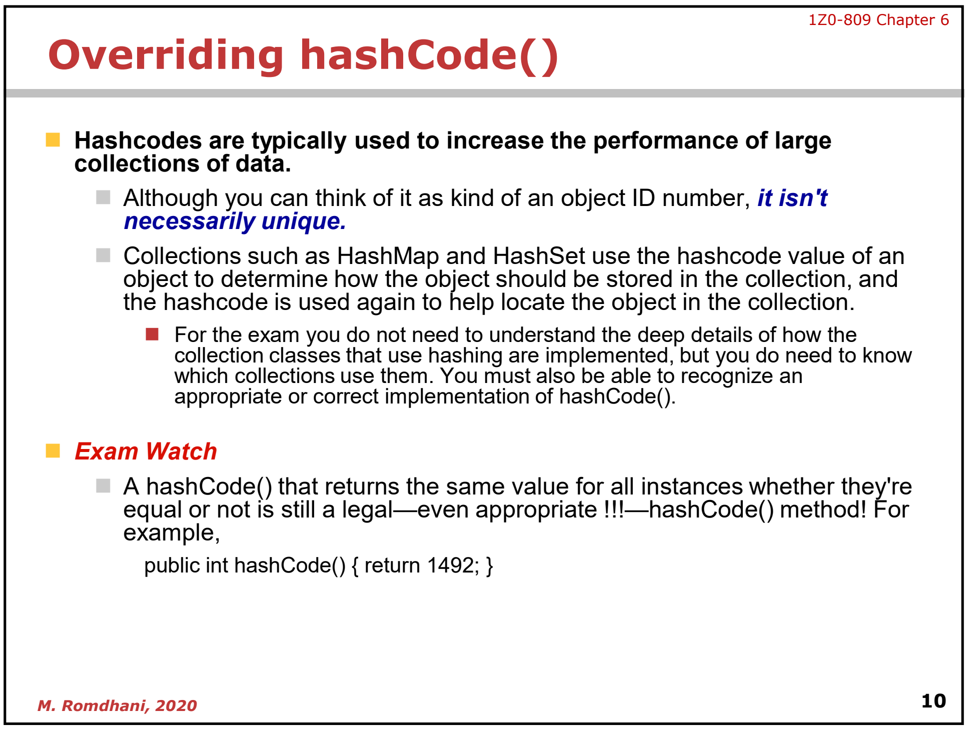
1Z0-809 Chapter 6

## Implementing an equals() Method

- **Exam Watch : Remember that the equals(), hashCode(), and toString() methods are all public.**
  - The following would not be a valid override of the equals() method, although it might appear to be if you don't look closely enough during the exam: class Foo { boolean equals(Object o) { } }
  - And watch out for the argument types as well. The following method is an overload, but not an override of the equals() method: class Boo { public boolean equals(Boo b) { } }
- **The equals() Contract**
  - equals() is reflexive. For any reference value x, x.equals(x) should return true.
  - It is symmetric. For any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
  - It is transitive. For any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.
  - It is consistent. For any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
  - For any non-null reference value x, x.equals(null) should return false.
  - ***If two objects are considered equal using the equals() method, then they must have identical hashCode() values***.

*M. Romdhani, 2020* 9

9



1Z0-809 Chapter 6

## Overriding hashCode()

- **Hashcodes are typically used to increase the performance of large collections of data.**
  - Although you can think of it as kind of an object ID number, ***it isn't necessarily unique.***
  - Collections such as HashMap and HashSet use the hashcode value of an object to determine how the object should be stored in the collection, and the hashcode is used again to help locate the object in the collection.
    - For the exam you do not need to understand the deep details of how the collection classes that use hashing are implemented, but you do need to know which collections use them. You must also be able to recognize an appropriate or correct implementation of hashCode().
- ***Exam Watch***
  - A hashCode() that returns the same value for all instances whether they're equal or not is still a legal—even appropriate !!!—hashCode() method! For example,

    public int hashCode() { return 1492; }

*M. Romdhani, 2020* 10

10

## Collections overview

11

## Key Interfaces and Classes of the Collections Framework

1Z0-809 Chapter 6

- **The Collections API begins with a group of interfaces, but also gives you a truckload of concrete classes.**
  - The core interfaces you need to know for the exam (and for life in general) are the following nine:

| | | |
|---|---|---|
| Collection | Set | SortedSet |
| List | Map | SortedMap |
| Queue | NavigableSet | NavigableMap |

  - Concrete Collection classes

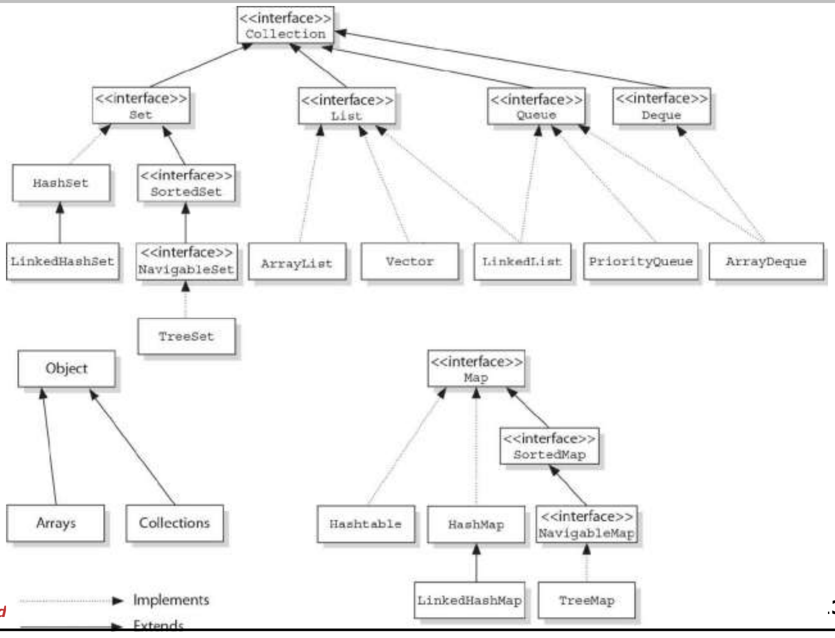
| Maps | Sets | Lists | Queues | Utilities |
|---|---|---|---|---|
| HashMap | HashSet | ArrayList | PriorityQueue | Collections |
| Hashtable | LinkedHashSet | Vector | ArrayDeque | Arrays |
| TreeMap | TreeSet | LinkedList | | |
| LinkedHashMap | | | | |

*M. Romdhani, 2020* 12

12

## The interface and class hierarchy for collections

1Z0-809 Chapter 6



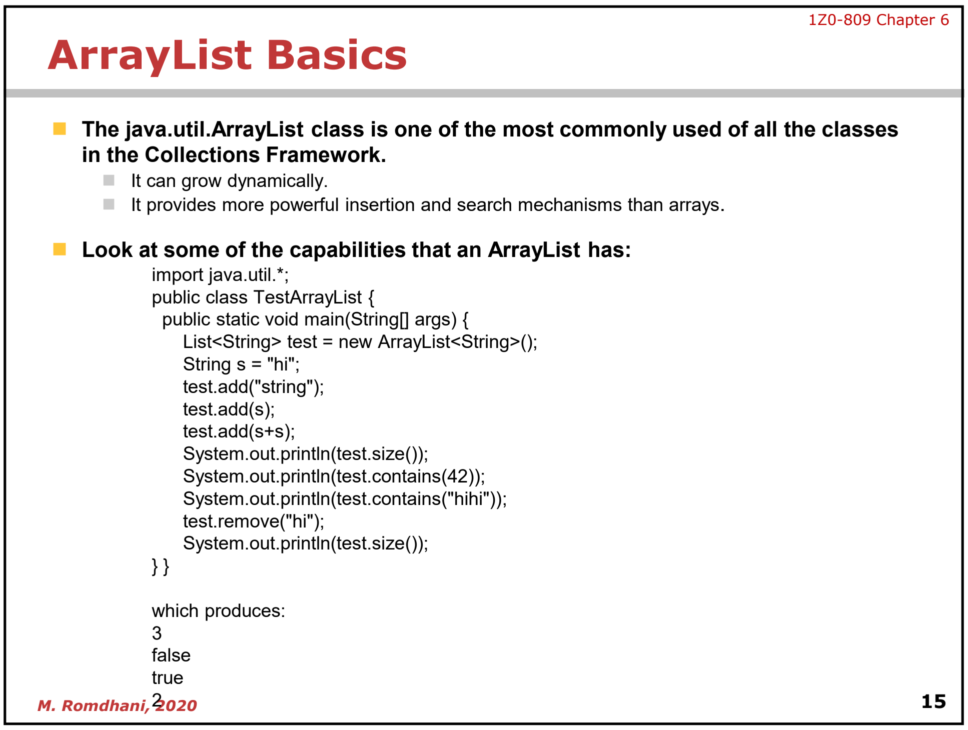
*M. Romd*

.3

13

## Using the Collections Framework

14



1Z0-809 Chapter 6

## ArrayList Basics

- **The java.util.ArrayList class is one of the most commonly used of all the classes in the Collections Framework.**
  - It can grow dynamically.
  - It provides more powerful insertion and search mechanisms than arrays.
- **Look at some of the capabilities that an ArrayList has:**

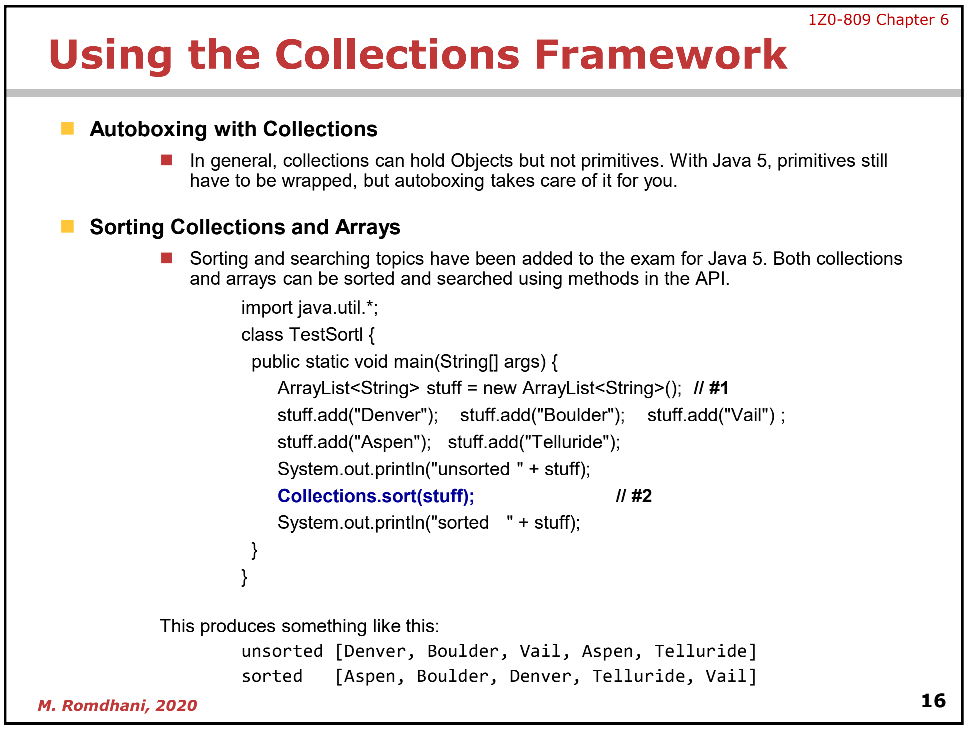
```
import java.util.*;
public class TestArrayList {
  public static void main(String[] args) {
      List<String> test = new ArrayList<String>();
      String s = "hi";
      test.add("string");
      test.add(s);
      test.add(s+s);
      System.out.println(test.size());
      System.out.println(test.contains(42));
      System.out.println(test.contains("hihi"));
      test.remove("hi");
      System.out.println(test.size());
} }
```

which produces:

```
3
false
true
2
```

*M. Romdhani, 2020* 15

15



1Z0-809 Chapter 6

## Using the Collections Framework

- **Autoboxing with Collections**
  - In general, collections can hold Objects but not primitives. With Java 5, primitives still have to be wrapped, but autoboxing takes care of it for you.
- **Sorting Collections and Arrays**
  - Sorting and searching topics have been added to the exam for Java 5. Both collections and arrays can be sorted and searched using methods in the API.

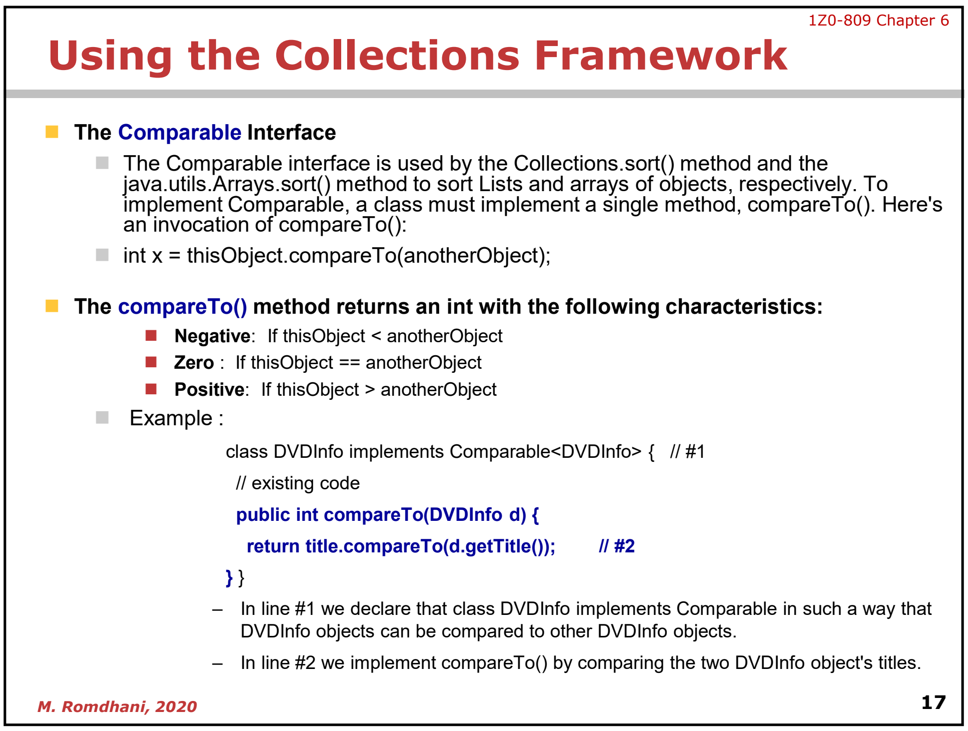
```
import java.util.*;
class TestSortl {
  public static void main(String[] args) {
     ArrayList<String> stuff = new ArrayList<String>();  // #1
     stuff.add("Denver");    stuff.add("Boulder");    stuff.add("Vail") ;
     stuff.add("Aspen");   stuff.add("Telluride");
     System.out.println("unsorted " + stuff);
     Collections.sort(stuff);                     // #2
     System.out.println("sorted   " + stuff);
  }
}
```

This produces something like this:

```
unsorted [Denver, Boulder, Vail, Aspen, Telluride]
sorted   [Aspen, Boulder, Denver, Telluride, Vail]
```

*M. Romdhani, 2020* 16

16



1Z0-809 Chapter 6

## Using the Collections Framework

- **The Comparable Interface**
  - The Comparable interface is used by the Collections.sort() method and the java.utils.Arrays.sort() method to sort Lists and arrays of objects, respectively. To implement Comparable, a class must implement a single method, compareTo(). Here's an invocation of compareTo():
  - int x = thisObject.compareTo(anotherObject);
- **The compareTo() method returns an int with the following characteristics:**
  - **Negative**: If thisObject < anotherObject
  - **Zero** : If thisObject == anotherObject
  - **Positive**: If thisObject > anotherObject
  - Example :

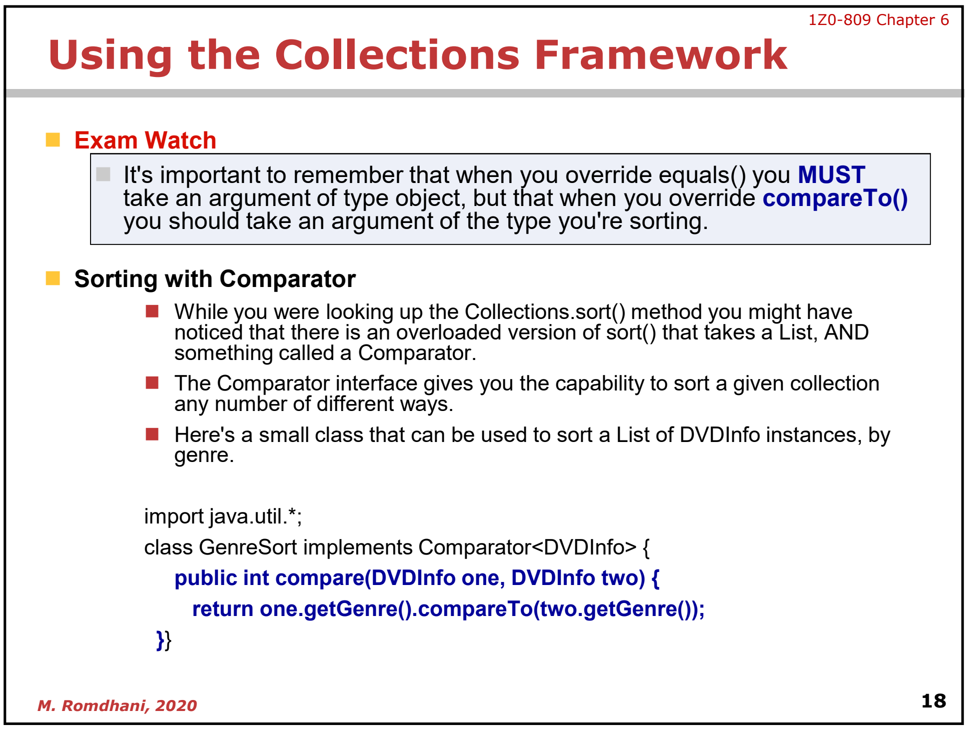
```
class DVDInfo implements Comparable<DVDInfo> {   // #1
  // existing code
  public int compareTo(DVDInfo d) {
    return title.compareTo(d.getTitle());       // #2
} }
```

– In line #1 we declare that class DVDInfo implements Comparable in such a way that DVDInfo objects can be compared to other DVDInfo objects.

– In line #2 we implement compareTo() by comparing the two DVDInfo object's titles.

*M. Romdhani, 2020* **17**

17



1Z0-809 Chapter 6

## Using the Collections Framework

- **Exam Watch**
  - It's important to remember that when you override equals() you **MUST** take an argument of type object, but that when you override **compareTo()** you should take an argument of the type you're sorting.
- **Sorting with Comparator**
  - While you were looking up the Collections.sort() method you might have noticed that there is an overloaded version of sort() that takes a List, AND something called a Comparator.
  - The Comparator interface gives you the capability to sort a given collection any number of different ways.
  - Here's a small class that can be used to sort a List of DVDInfo instances, by genre.

```
import java.util.*;
class GenreSort implements Comparator<DVDInfo> {
    public int compare(DVDInfo one, DVDInfo two) {
       return one.getGenre().compareTo(two.getGenre());
 }}
```

*M. Romdhani, 2020* **18**

18

## Creating a Comparator with a Lambda Expression

- **Let's take another look at the GenreSort class:**

```
class GenreSort implements Comparator<DVDInfo> {
   public int compare(DVDInfo one, DVDInfo two) {
      return one.getGenre().compareTo(two.getGenre());
   }
}
```

  - Because GenreSort is implementing a **functional interface**, we can actually replace the entire class with a lambda expression.

- **Now we have a lambda expression for the Comparator, so how do we use it? All we have to do is replace the GenreSort instance in our code with the lambda:**

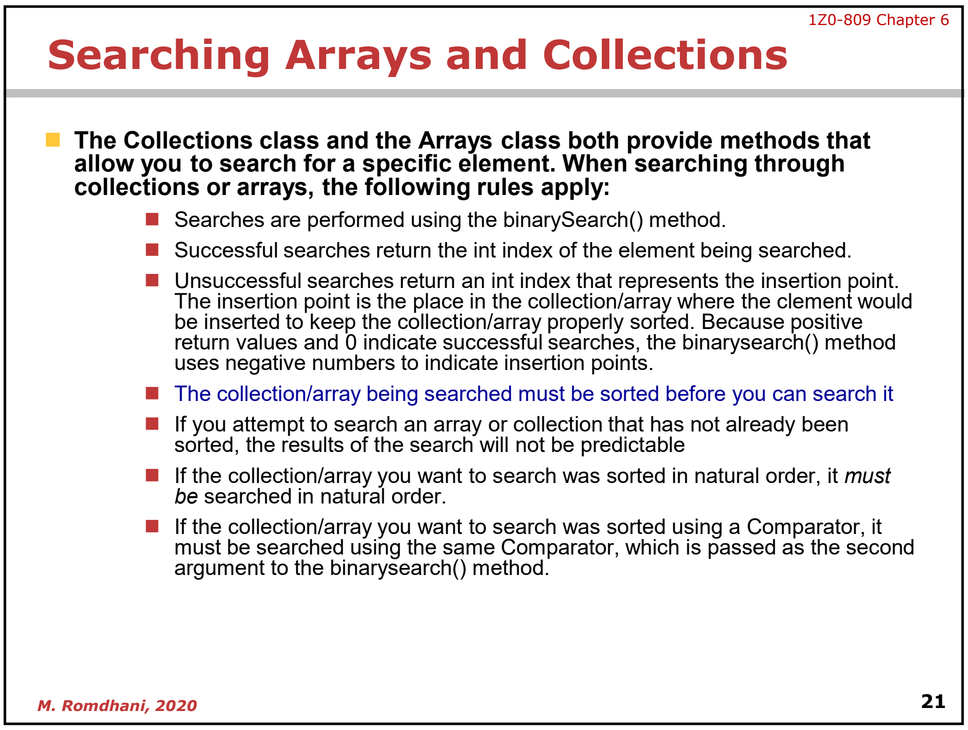
```
Collections.sort(dvdlist,
  (one, two) -> one.getGenre().compareTo(two.getGenre()));
```

## Comparing Comparable to Comparator

- **Because the Comparable and Comparator interfaces are so similar, expect the exam to try to confuse you**
  - For instance you might be asked to implement the compareTo() method in the Comparator interface.

| java.lang.Comparable | java.util.Comparator |
|---|---|
| `int objOne.compareTo(objTwo)` | `int compare(objone, objTwo)` |
| Returns<br>negative if `obj One < objTwo`<br>zero if `objOne == objTwo`<br>positive if `objOne > objTwo` | Same as Comparable |
| You must modify the class whose instances you want to sort. | You build a class separate from the class whose instances you want to sort. |
| Only **one** sort sequence can be created | **Many** sort sequences can be created |
| Implemented frequently in the API by: String, Wrapper classes, Date, Calendar... | Meant to be implemented to sort instances of third-party classes. |

- **Sorting with the Arrays Class**
  - The Arrays.sort() method is overridden in the same way the Collections.sort() method is.
    - Arrays.sort(arrayToSort)
    - **Arrays.sort(arrayToSort, Comparator)**



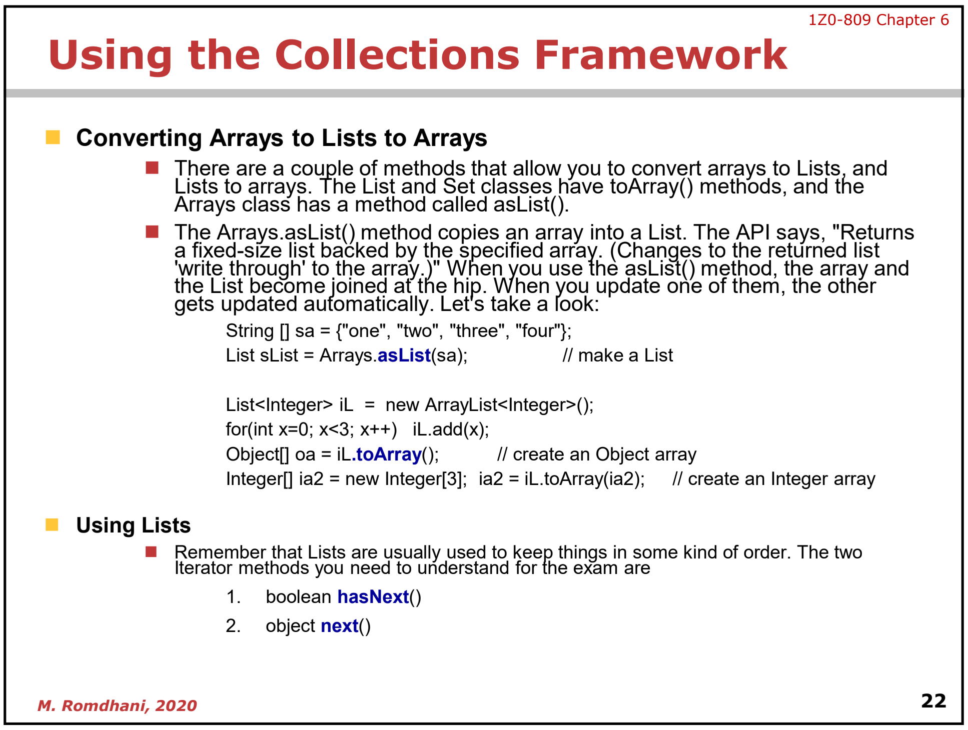
1Z0-809 Chapter 6

## Searching Arrays and Collections

- **The Collections class and the Arrays class both provide methods that allow you to search for a specific element. When searching through collections or arrays, the following rules apply:**
  - Searches are performed using the binarySearch() method.
  - Successful searches return the int index of the element being searched.
  - Unsuccessful searches return an int index that represents the insertion point. The insertion point is the place in the collection/array where the clement would be inserted to keep the collection/array properly sorted. Because positive return values and 0 indicate successful searches, the binarysearch() method uses negative numbers to indicate insertion points.
  - The collection/array being searched must be sorted before you can search it
  - If you attempt to search an array or collection that has not already been sorted, the results of the search will not be predictable
  - If the collection/array you want to search was sorted in natural order, it *must be* searched in natural order.
  - If the collection/array you want to search was sorted using a Comparator, it must be searched using the same Comparator, which is passed as the second argument to the binarysearch() method.

*M. Romdhani, 2020* **21**

21



1Z0-809 Chapter 6

## Using the Collections Framework

- **Converting Arrays to Lists to Arrays**
  - There are a couple of methods that allow you to convert arrays to Lists, and Lists to arrays. The List and Set classes have toArray() methods, and the Arrays class has a method called asList().
  - The Arrays.asList() method copies an array into a List. The API says, "Returns a fixed-size list backed by the specified array. (Changes to the returned list 'write through' to the array.)" When you use the asList() method, the array and the List become joined at the hip. When you update one of them, the other gets updated automatically. Let's take a look:

```
String [] sa = {"one", "two", "three", "four"};
List sList = Arrays.asList(sa);                // make a List

List<Integer> iL  =  new ArrayList<Integer>();
for(int x=0; x<3; x++)   iL.add(x);
Object[] oa = iL.toArray();          // create an Object array
Integer[] ia2 = new Integer[3];  ia2 = iL.toArray(ia2);     // create an Integer array
```

- **Using Lists**
  - Remember that Lists are usually used to keep things in some kind of order. The two Iterator methods you need to understand for the exam are
    1. boolean **hasNext**()
    2. object **next**()

*M. Romdhani, 2020* **22**

22

1Z0-809 Chapter 6

## Using the Collections Framework

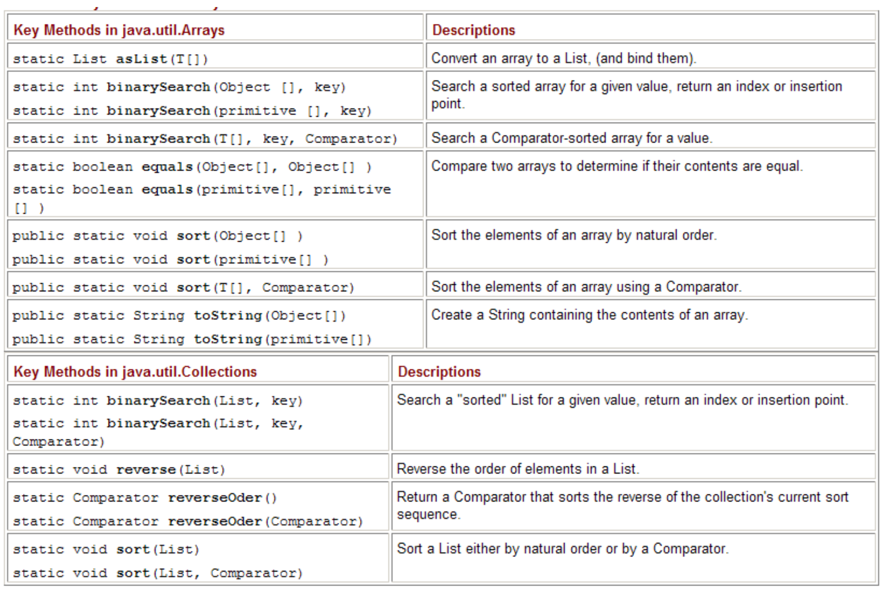
- **Using Sets**
  - Remember that Sets are used when you don't want any duplicates in your collection.
  - If you attempt to add an element to a set that already exists in the set, the duplicate element will not be added, and the add() method will return false.
  - Remember, HashSets tend to be very fast because, as we discussed earlier, they use hashcodes.
- **Using Maps**
  - Remember that when you use a class that implements Map, any classes that you use as a part of the keys for that map must override the hashCode() and equals() methods.

*M. Romdhani, 2020* **23**

23

1Z0-809 Chapter 6

## Key Methods in Arrays and Collections



| Key Methods in java.util.Arrays | Descriptions |
|---|---|
| `static List asList(T[])` | Convert an array to a List, (and bind them). |
| `static int binarySearch(Object [], key)`<br>`static int binarySearch(primitive [], key)` | Search a sorted array for a given value, return an index or insertion point. |
| `static int binarySearch(T[], key, Comparator)` | Search a Comparator-sorted array for a value. |
| `static boolean equals(Object[], Object[] )`<br>`static boolean equals(primitive[], primitive [] )` | Compare two arrays to determine if their contents are equal. |
| `public static void sort(Object[] )`<br>`public static void sort(primitive[] )` | Sort the elements of an array by natural order. |
| `public static void sort(T[], Comparator)` | Sort the elements of an array using a Comparator. |
| `public static String toString(Object[])`<br>`public static String toString(primitive[])` | Create a String containing the contents of an array. |

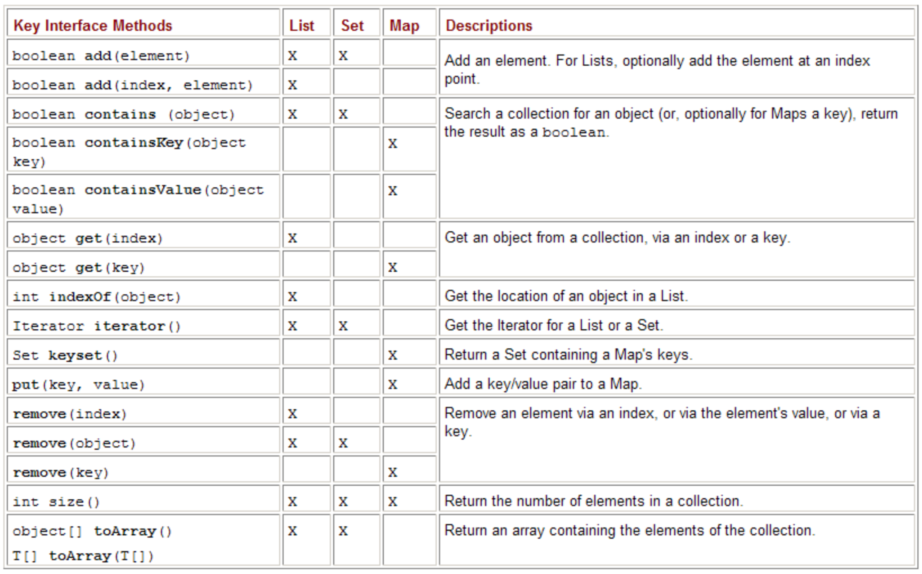
| Key Methods in java.util.Collections | Descriptions |
|---|---|
| `static int binarySearch(List, key)`<br>`static int binarySearch(List, key, Comparator)` | Search a "sorted" List for a given value, return an index or insertion point. |
| `static void reverse(List)` | Reverse the order of elements in a List. |
| `static Comparator reverseOder()`<br>`static Comparator reverseOder(Comparator)` | Return a Comparator that sorts the reverse of the collection's current sort sequence. |
| `static void sort(List)`<br>`static void sort(List, Comparator)` | Sort a List either by natural order or by a Comparator. |

*M. Romdhani, 2020* **24**

24

1Z0-809 Chapter 6

## Key Methods in List, Set, and Map



| Key Interface Methods | List | Set | Map | Descriptions |
|---|---|---|---|---|
| `boolean add(element)` | X | X | | Add an element. For Lists, optionally add the element at an index point. |
| `boolean add(index, element)` | X | | | |
| `boolean contains (object)` | X | X | | Search a collection for an object (or, optionally for Maps a key), return the result as a `boolean`. |
| `boolean containsKey(object key)` | | | X | |
| `boolean containsValue(object value)` | | | X | |
| `object get(index)` | X | | | Get an object from a collection, via an index or a key. |
| `object get(key)` | | | X | |
| `int indexOf(object)` | X | | | Get the location of an object in a List. |
| `Iterator iterator()` | X | X | | Get the Iterator for a List or a Set. |
| `Set keyset()` | | | X | Return a Set containing a Map's keys. |
| `put(key, value)` | | | X | Add a key/value pair to a Map. |
| `remove(index)` | X | | | Remove an element via an index, or via the element's value, or via a key. |
| `remove(object)` | X | X | | |
| `remove(key)` | | | X | |
| `int size()` | X | X | X | Return the number of elements in a collection. |
| `object[] toArray()` `T[] toArray(T[])` | X | X | | Return an array containing the elements of the collection. |

*M. Romdhani, 2020* **25**

25

1Z0-809 Chapter 6

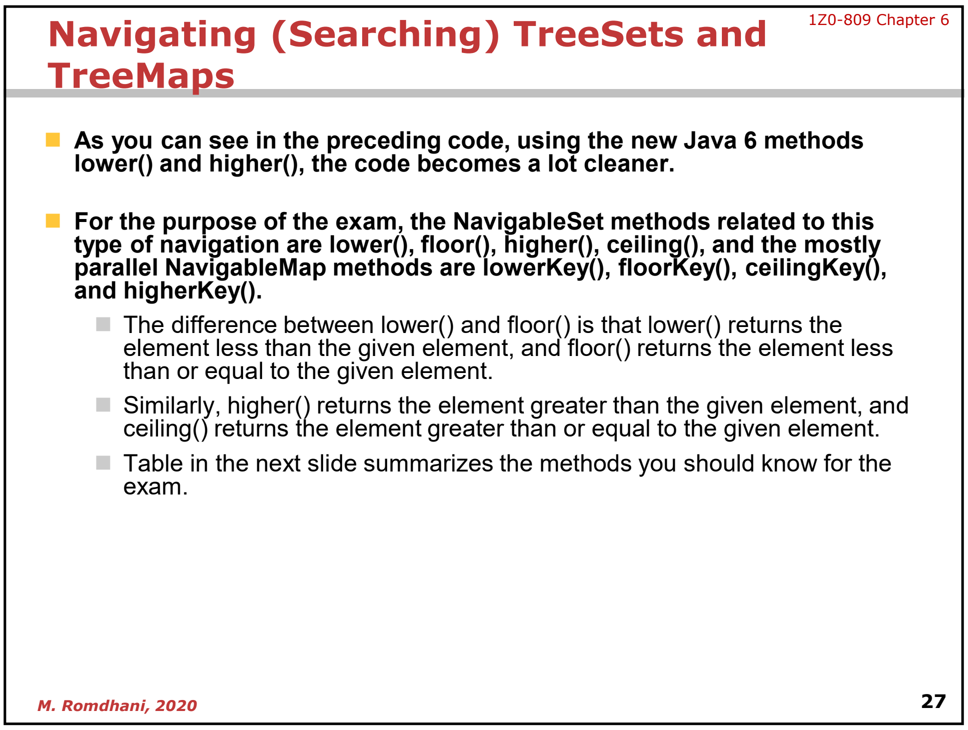
## Navigating (Searching) TreeSets and TreeMaps

- **Java 6 introduced (among others) two new interfaces: java.util.NavigableSet and java.util.NavigableMap.**
  - For the purposes of the exam, you're interested in how **TreeSet** and **TreeMap** implement these interfaces.

```
import java.util.*;
public class Ferry {
public static void main(String[] args) {
TreeSet<Integer> times = new TreeSet<Integer>();
times.add(1205); // add some departure times
times.add(1505);     times.add(1545);     times.add(1830);
 times.add(2010);     times.add(2100);
// Java 5 version
TreeSet<Integer> subset = new TreeSet<Integer>();
subset = (TreeSet)times.headSet(1600);
System.out.println("J5 - last before 4pm is: " +
                                          subset.last());
TreeSet<Integer> sub2 = new TreeSet<Integer>();
sub2 = (TreeSet)times.tailSet(2000);
System.out.println("J5 - first after 8pm is: " + sub2.first());
// Java 6 version using the new lower() and higher() methods
System.out.println("J6 - last before 4pm is: " +
                                           times.lower(1600));
System.out.println("J6 - first after 8pm is: " +
                                        times.higher(2000));
}}
```

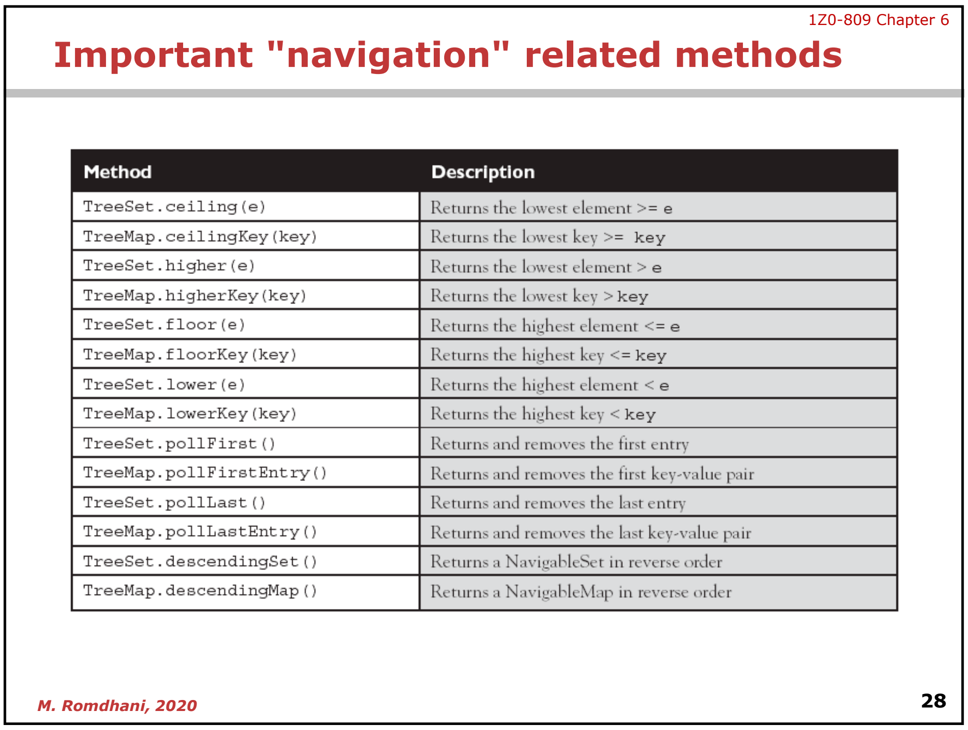
*M. Romdhani, 2020* **26**

26



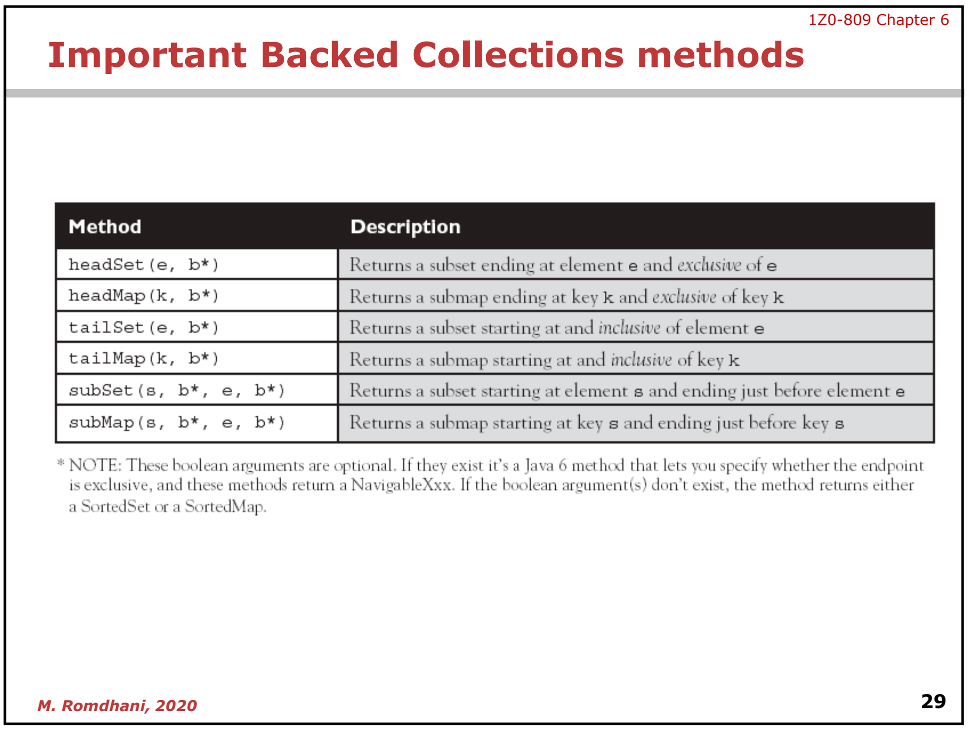
## Navigating (Searching) TreeSets and TreeMaps

- **As you can see in the preceding code, using the new Java 6 methods lower() and higher(), the code becomes a lot cleaner.**
- **For the purpose of the exam, the NavigableSet methods related to this type of navigation are lower(), floor(), higher(), ceiling(), and the mostly parallel NavigableMap methods are lowerKey(), floorKey(), ceilingKey(), and higherKey().**
  - The difference between lower() and floor() is that lower() returns the element less than the given element, and floor() returns the element less than or equal to the given element.
  - Similarly, higher() returns the element greater than the given element, and ceiling() returns the element greater than or equal to the given element.
  - Table in the next slide summarizes the methods you should know for the exam.



## Important "navigation" related methods

| Method | Description |
|---|---|
| `TreeSet.ceiling(e)` | Returns the lowest element >= `e` |
| `TreeMap.ceilingKey(key)` | Returns the lowest key >= `key` |
| `TreeSet.higher(e)` | Returns the lowest element > `e` |
| `TreeMap.higherKey(key)` | Returns the lowest key > `key` |
| `TreeSet.floor(e)` | Returns the highest element <= `e` |
| `TreeMap.floorKey(key)` | Returns the highest key <= `key` |
| `TreeSet.lower(e)` | Returns the highest element < `e` |
| `TreeMap.lowerKey(key)` | Returns the highest key < `key` |
| `TreeSet.pollFirst()` | Returns and removes the first entry |
| `TreeMap.pollFirstEntry()` | Returns and removes the first key-value pair |
| `TreeSet.pollLast()` | Returns and removes the last entry |
| `TreeMap.pollLastEntry()` | Returns and removes the last key-value pair |
| `TreeSet.descendingSet()` | Returns a NavigableSet in reverse order |
| `TreeMap.descendingMap()` | Returns a NavigableMap in reverse order |



1Z0-809 Chapter 6

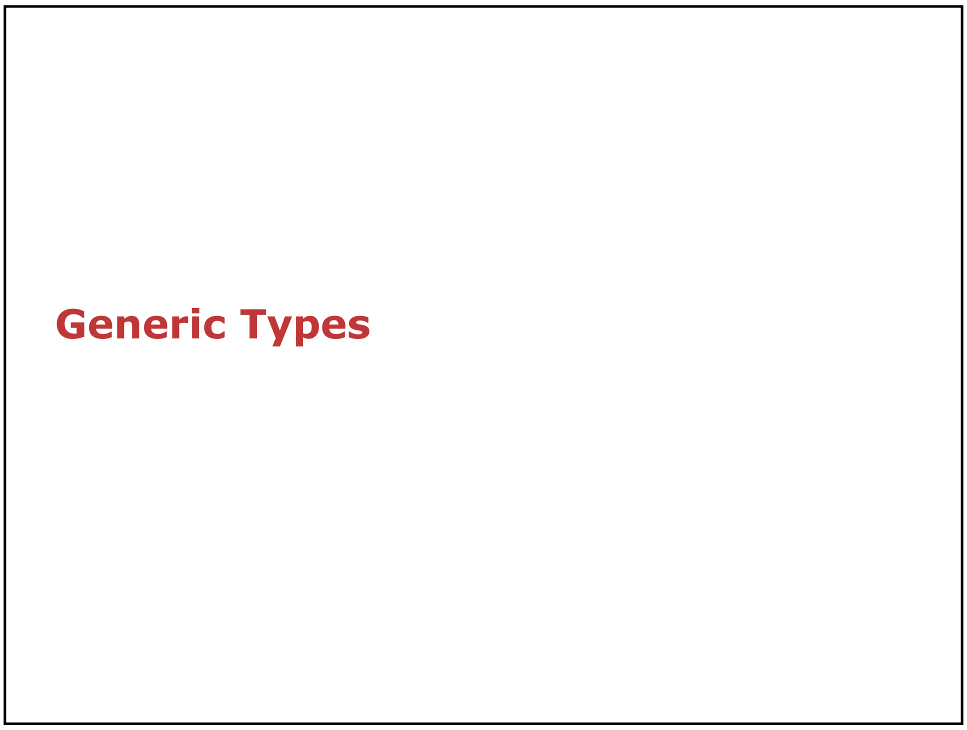
## Important Backed Collections methods

| Method | Description |
|---|---|
| `headSet(e, b*)` | Returns a subset ending at element `e` and *exclusive* of `e` |
| `headMap(k, b*)` | Returns a submap ending at key `k` and *exclusive* of key `k` |
| `tailSet(e, b*)` | Returns a subset starting at and *inclusive* of element `e` |
| `tailMap(k, b*)` | Returns a submap starting at and *inclusive* of key `k` |
| `subSet(s, b*, e, b*)` | Returns a subset starting at element `s` and ending just before element `e` |
| `subMap(s, b*, e, b*)` | Returns a submap starting at key `s` and ending just before key `s` |

* NOTE: These boolean arguments are optional. If they exist it's a Java 6 method that lets you specify whether the endpoint is exclusive, and these methods return a NavigableXxx. If the boolean argument(s) don't exist, the method returns either a SortedSet or a SortedMap.

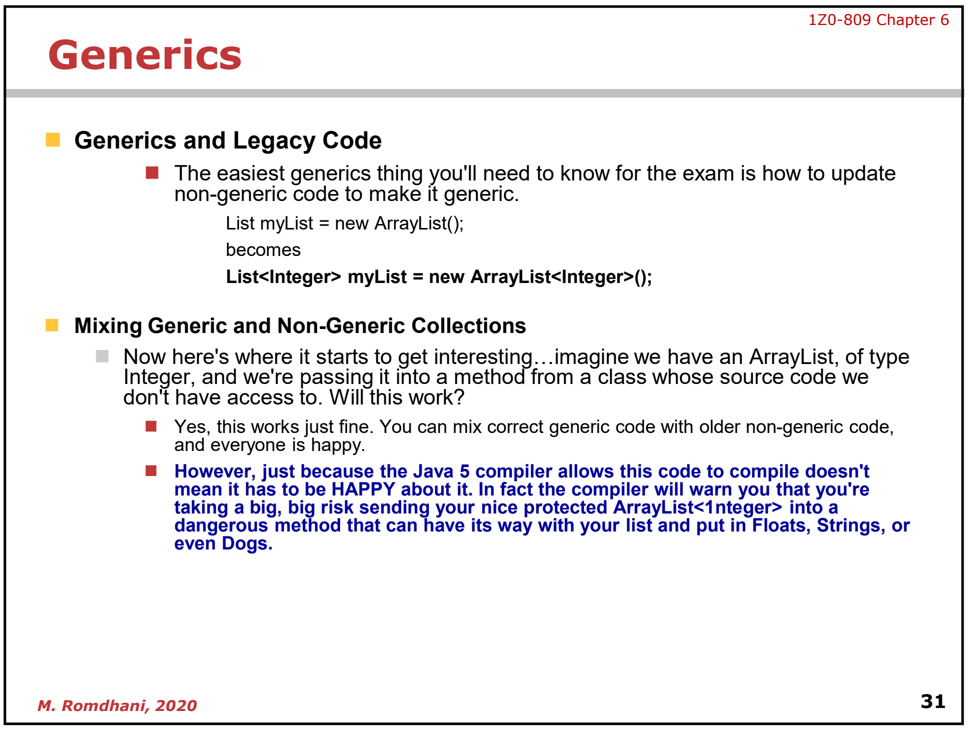
*M. Romdhani, 2020* 29

29



## Generic Types

30



## Generics

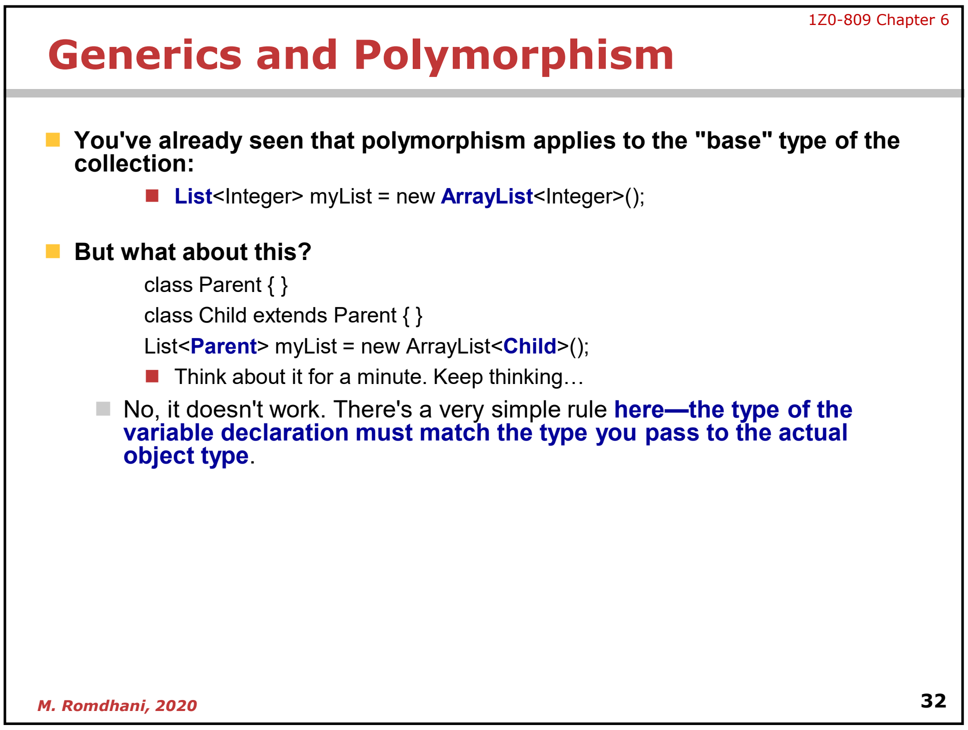
- **Generics and Legacy Code**
  - The easiest generics thing you'll need to know for the exam is how to update non-generic code to make it generic.

    List myList = new ArrayList();

    becomes

    **List<Integer> myList = new ArrayList<Integer>();**

- **Mixing Generic and Non-Generic Collections**
  - Now here's where it starts to get interesting…imagine we have an ArrayList, of type Integer, and we're passing it into a method from a class whose source code we don't have access to. Will this work?
    - Yes, this works just fine. You can mix correct generic code with older non-generic code, and everyone is happy.
    - **However, just because the Java 5 compiler allows this code to compile doesn't mean it has to be HAPPY about it. In fact the compiler will warn you that you're taking a big, big risk sending your nice protected ArrayList<1nteger> into a dangerous method that can have its way with your list and put in Floats, Strings, or even Dogs.**



## Generics and Polymorphism

- **You've already seen that polymorphism applies to the "base" type of the collection:**
  - **List**<Integer> myList = new **ArrayList**<Integer>();

- **But what about this?**

  class Parent { }

  class Child extends Parent { }

  List<**Parent**> myList = new ArrayList<**Child**>();
    - Think about it for a minute. Keep thinking…
  - No, it doesn't work. There's a very simple rule **here—the type of the variable declaration must match the type you pass to the actual object type**.

## Generic Methods

- **The Rule :**
  - One way to remember this is that if you see the wildcard notation (a question mark ?), this means "many possibilities".
  - If you do NOT see the question mark, then it means the <type> in the brackets, and absolutely NOTHING ELSE. List<Dog> means List<Dog> and not List<Beagle>, List<Poodle>, or any other subtype of Dog. But List<? extends Dog> could mean List<Beagle>, List<Poodle>, and so on. Of course List<?> could be… anything at all.
- **Exercise : look at the following statements and figure out which will compile:**

  1) List<?> list = new ArrayList<Dog>();
  2) List<? extends Animal> aList = new ArrayList<Dog>();
  3) List<?> foo = new ArrayList<? extends Animal>();
  4) List<? extends Dog> cList = new ArrayList<Integer>();
  5) List<? super Dog> bList = new ArrayList<Animal>();
  6) List<? super Animal> dList = new ArrayList<Dog>();

  The correct answers (the statements that compile) are 1, 2, and 5. The three that won't compile are
  - Statement: List<?> foo = new ArrayList<? extends Animal>();
    - Problem: you cannot use wildcard notation in the object creation. So the new ArrayList<? extends Animal>() will not compile.
  - Statement: List<? extends Dog> cList = new ArrayList<Integer>();
    - Problem: You cannot assign an Integer list to a reference that takes only a Dog (including any subtypes of Dog, of course).
  - Statement: List<? super Animal> dList = new ArrayList<Dog>();
    - Problem: You cannot assign a Dog to <? super Animal>. The Dog is too "low" in the class hierarchy. Only <Animal> or <Object> would have been legal.

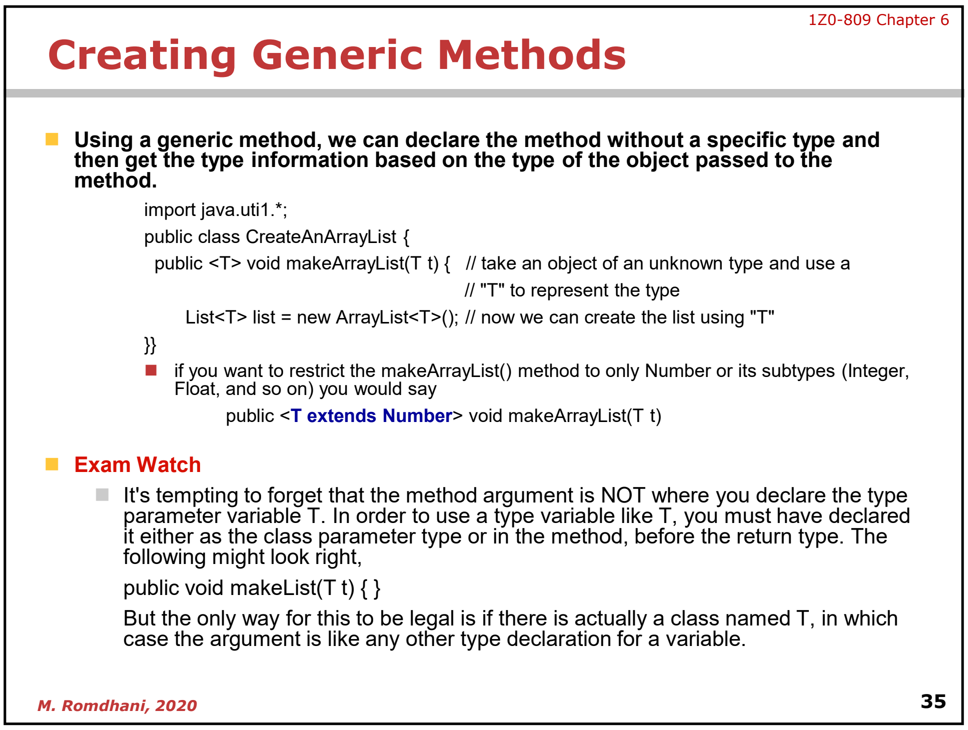
## Making Your Own Generic Class

- **Parameterized type of the class :**

```
public class TestGenerics<T> {  // as the class type
    T anInstance;              // as an instance variable type
    T [ ] anArrayOfTs;          // as an array type
    TestGenerics(T anInstance) {     // as an argument type
        this.anInstance = anInstance;
     }
    T getT() {                 // as a return type
        return anInstance;
  }}
```

- **You can use a form of wildcard notation in a class definition**

```
public class AnimalHolder<T extends Animal> { // use "T" instead of "?"
   public static void main(String[] args) {
      AnimalHolder<Dog> dogHolder = new AnimalHolder<Dog>(); // OK
      AnimalHolder<Integer> x = new AnimalHolder<Integer>(); // NO!
   }
}
```



1Z0-809 Chapter 6

## Creating Generic Methods

- **Using a generic method, we can declare the method without a specific type and then get the type information based on the type of the object passed to the method.**

```
import java.uti1.*;
public class CreateAnArrayList {
  public <T> void makeArrayList(T t) {   // take an object of an unknown type and use a
                                         // "T" to represent the type
      List<T> list = new ArrayList<T>(); // now we can create the list using "T"
}}
```

  - if you want to restrict the makeArrayList() method to only Number or its subtypes (Integer, Float, and so on) you would say

    public <**T extends Number**> void makeArrayList(T t)

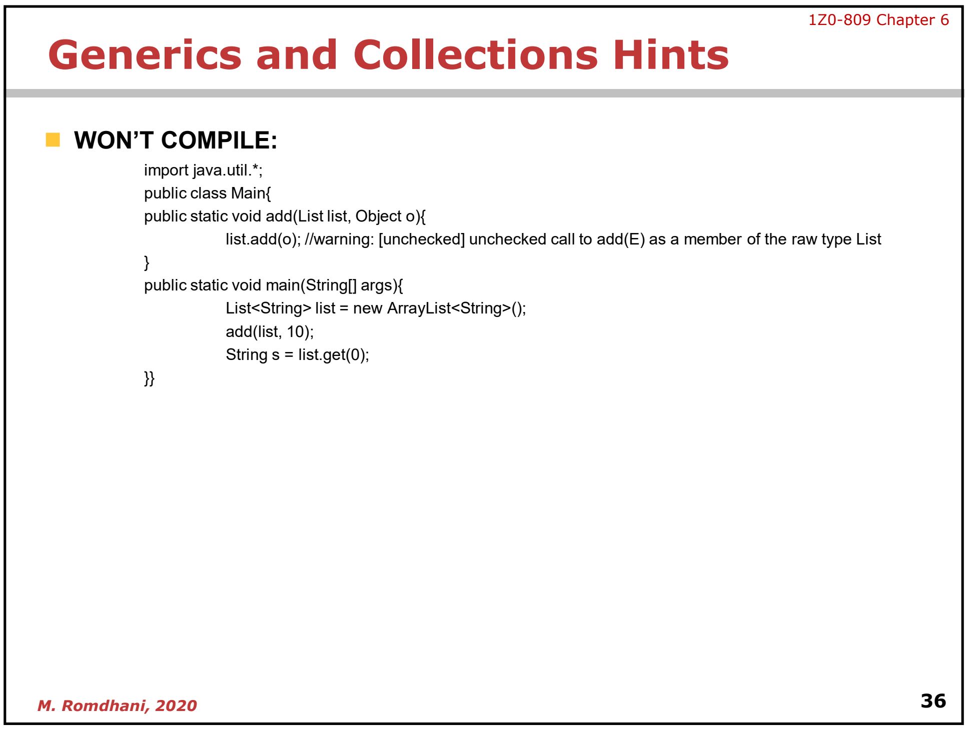
- **Exam Watch**
  - It's tempting to forget that the method argument is NOT where you declare the type parameter variable T. In order to use a type variable like T, you must have declared it either as the class parameter type or in the method, before the return type. The following might look right,

    public void makeList(T t) { }

    But the only way for this to be legal is if there is actually a class named T, in which case the argument is like any other type declaration for a variable.

*M. Romdhani, 2020* **35**

35



1Z0-809 Chapter 6

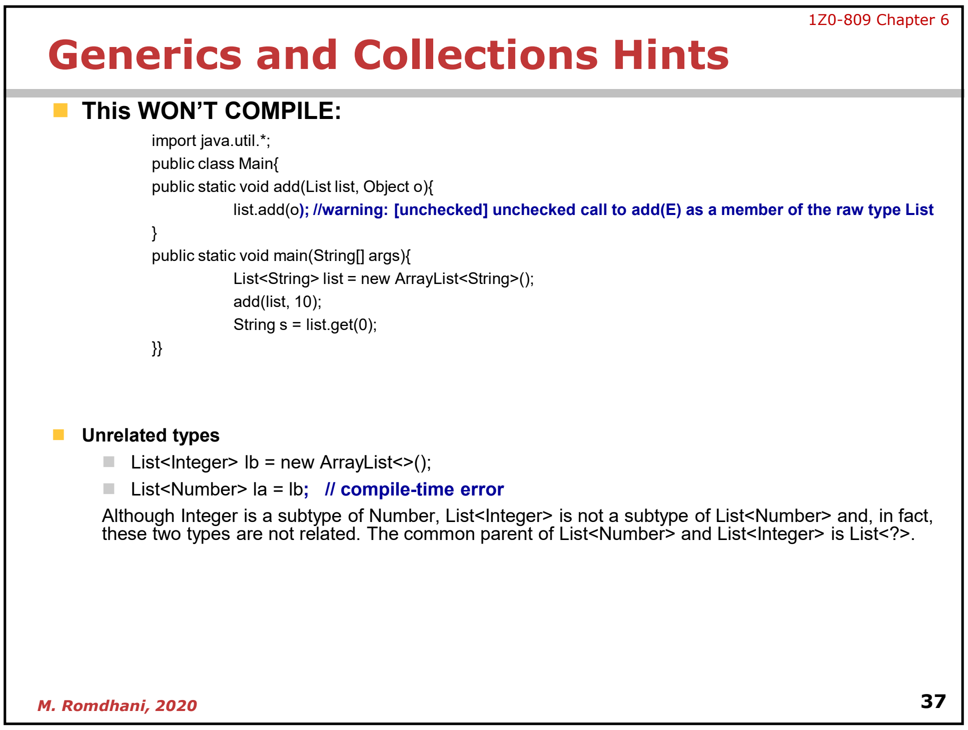
## Generics and Collections Hints

- **WON'T COMPILE:**

```
import java.util.*;
public class Main{
public static void add(List list, Object o){
        list.add(o); //warning: [unchecked] unchecked call to add(E) as a member of the raw type List
}
public static void main(String[] args){
        List<String> list = new ArrayList<String>();
        add(list, 10);
        String s = list.get(0);
}}
```

*M. Romdhani, 2020* **36**

36



1Z0-809 Chapter 6

## Generics and Collections Hints

- **This WON'T COMPILE:**

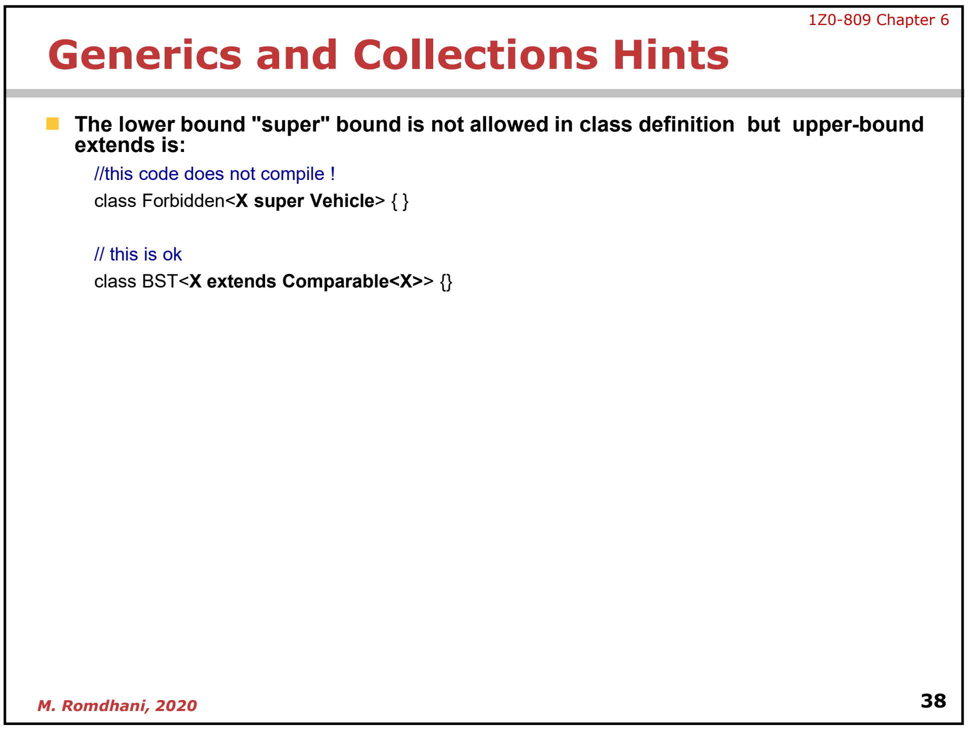
```
import java.util.*;
public class Main{
public static void add(List list, Object o){
        list.add(o); //warning: [unchecked] unchecked call to add(E) as a member of the raw type List
}
public static void main(String[] args){
        List<String> list = new ArrayList<String>();
        add(list, 10);
        String s = list.get(0);
}}
```

- **Unrelated types**
  - List<Integer> lb = new ArrayList<>();
  - List<Number> la = lb;   **// compile-time error**

  Although Integer is a subtype of Number, List<Integer> is not a subtype of List<Number> and, in fact, these two types are not related. The common parent of List<Number> and List<Integer> is List<?>.

***M. Romdhani, 2020*** **37**

37



1Z0-809 Chapter 6

## Generics and Collections Hints

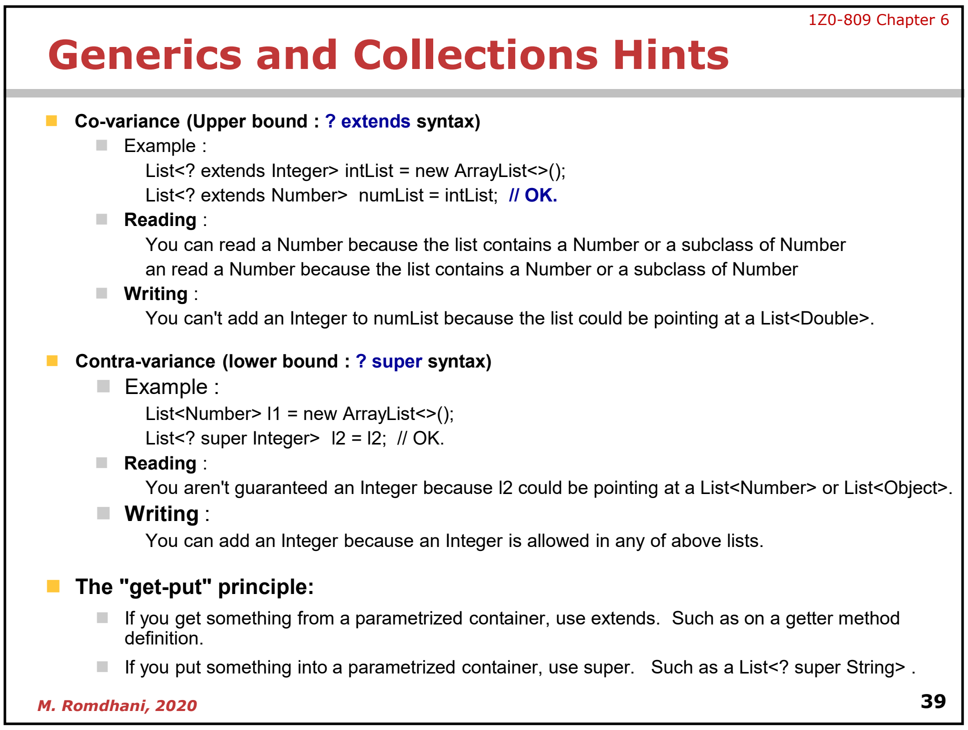
- **The lower bound "super" bound is not allowed in class definition  but  upper-bound extends is:**

```
//this code does not compile !
class Forbidden<X super Vehicle> { }

// this is ok
class BST<X extends Comparable<X>> {}
```

***M. Romdhani, 2020*** **38**

38



1Z0-809 Chapter 6

## Generics and Collections Hints

- **Co-variance (Upper bound : ? extends syntax)**
  - Example :
    List<? extends Integer> intList = new ArrayList<>();
    List<? extends Number>  numList = intList;  **// OK.**
  - **Reading** :
    You can read a Number because the list contains a Number or a subclass of Number
    an read a Number because the list contains a Number or a subclass of Number
  - **Writing** :
    You can't add an Integer to numList because the list could be pointing at a List<Double>.
- **Contra-variance (lower bound : ? super syntax)**
  - Example :
    List<Number> l1 = new ArrayList<>();
    List<? super Integer>  l2 = l2;  // OK.
  - **Reading** :
    You aren't guaranteed an Integer because l2 could be pointing at a List<Number> or List<Object>.
  - **Writing** :
    You can add an Integer because an Integer is allowed in any of above lists.
- **The "get-put" principle:**
  - If you get something from a parametrized container, use extends.  Such as on a getter method definition.
  - If you put something into a parametrized container, use super.   Such as a List<? super String> .

*M. Romdhani, 2020* **39**

39