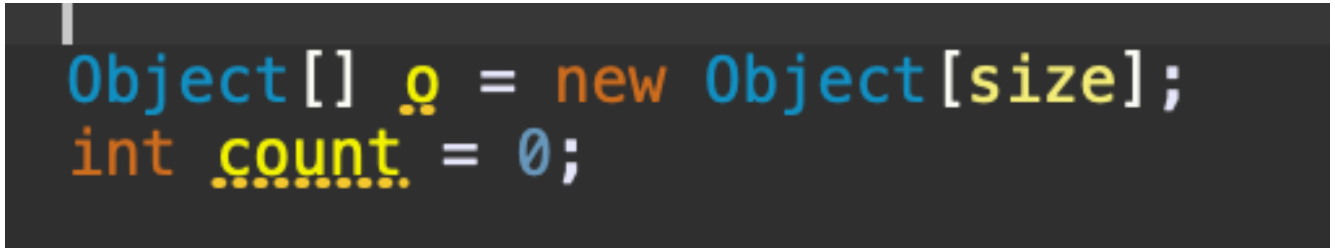
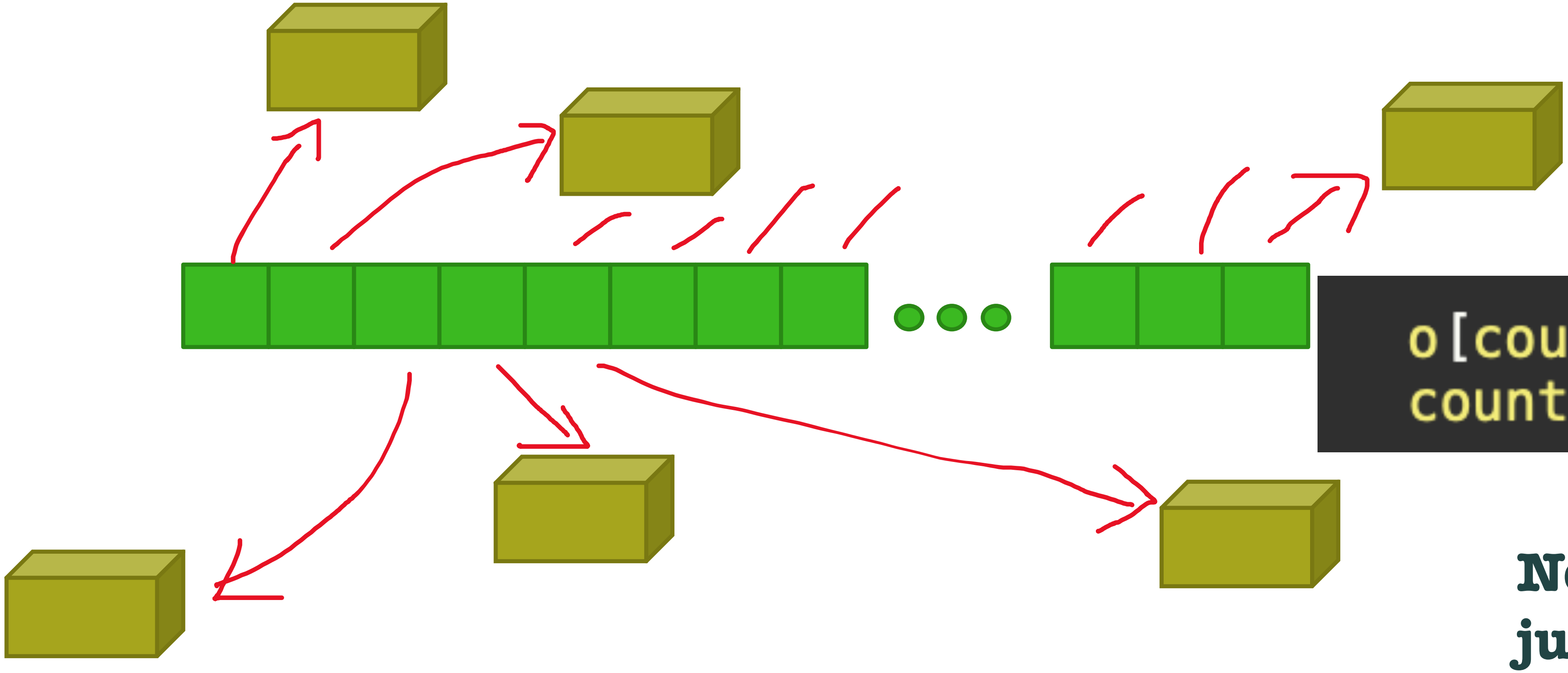
COLLECTIONS II

WEEK 3 PRESENTATION 3 - SEARCHING ARRAYS



```
Object[] o = new Object[size];
int count = 0;
```



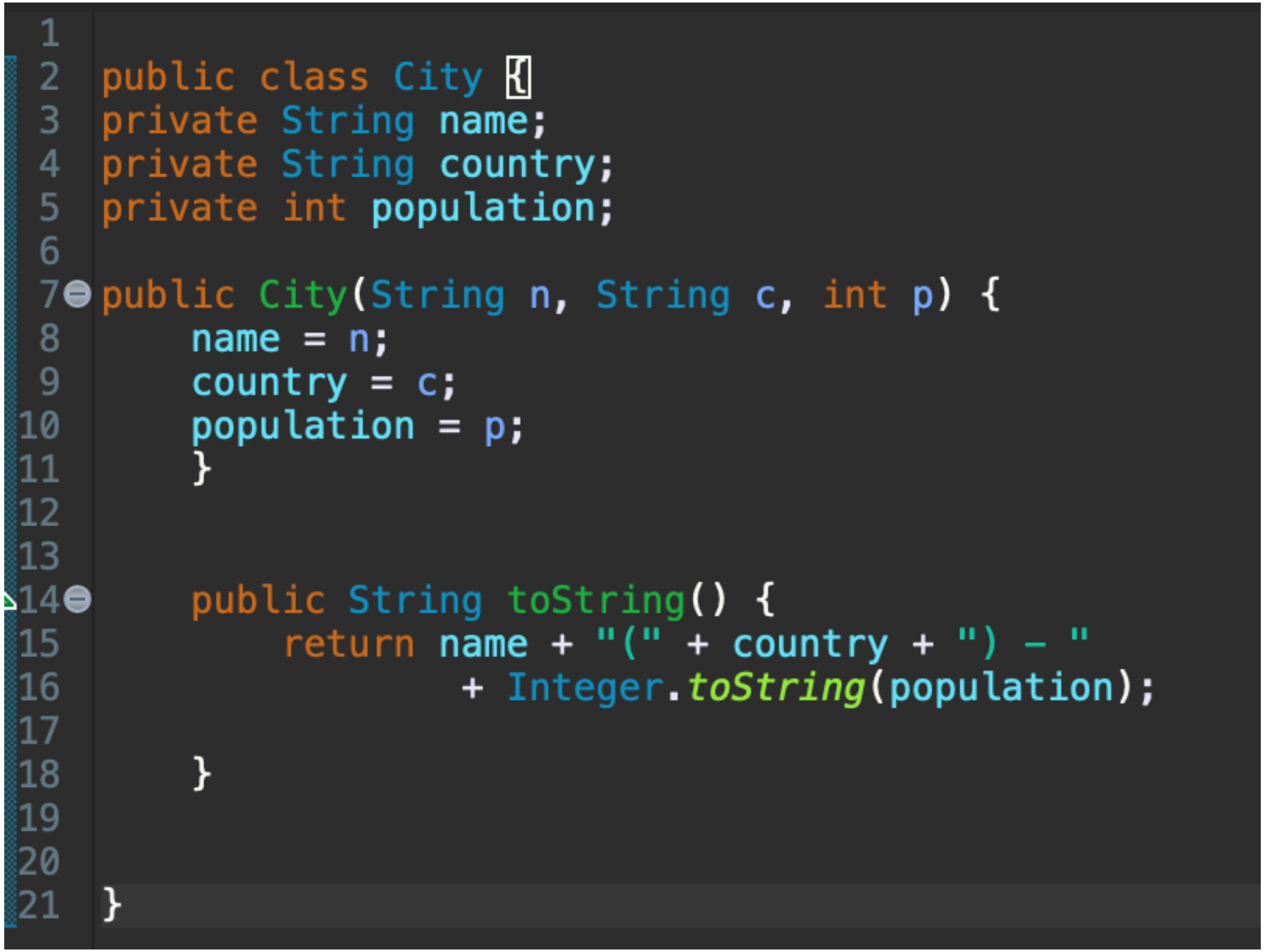
```
o[count] = new Object();
count++;
```

The simplest group of objects is just an array of objects.

Note: initially its just an array of empty references

SEARCHING COLLECTIONS OF OBJECTS

- Common operations that involve searching through groups of objects
- contains (Object obj)
  - True if the group contains the object
- indexOf(Object obj)
  - Returns the index of the first occurrence of the object
  - See also lastIndexOf
- remove(Object obj)
  - Removes object from the collection

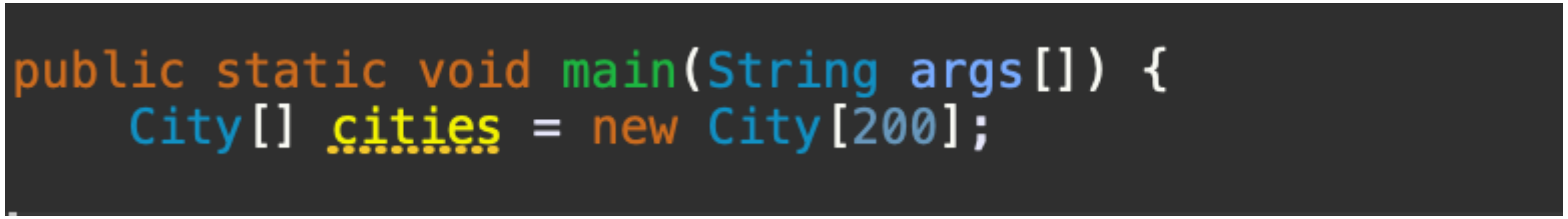


```java
public class City {
private String name;
private String country;
private int population;

public City(String n, String c, int p) {
    name = n;
    country = c;
    population = p;
    }


    public String toString() {
        return name + "(" + country + ") - "
                + Integer.toString(population);

    }


}
```

Suppose that we have an array of City objects…

```
#name, country, population
Shanghai,cn,24256800
Karachi,pk,23500000
Beijing,cn,21516000
Dhaka,bd,16970105
Delhi,in,16787941
Lagos,ng,16060303
Istanbul,tr,14025000
Tokyo,jp,13513734
Guangzhou,cn,13080500
…
```

Create an array

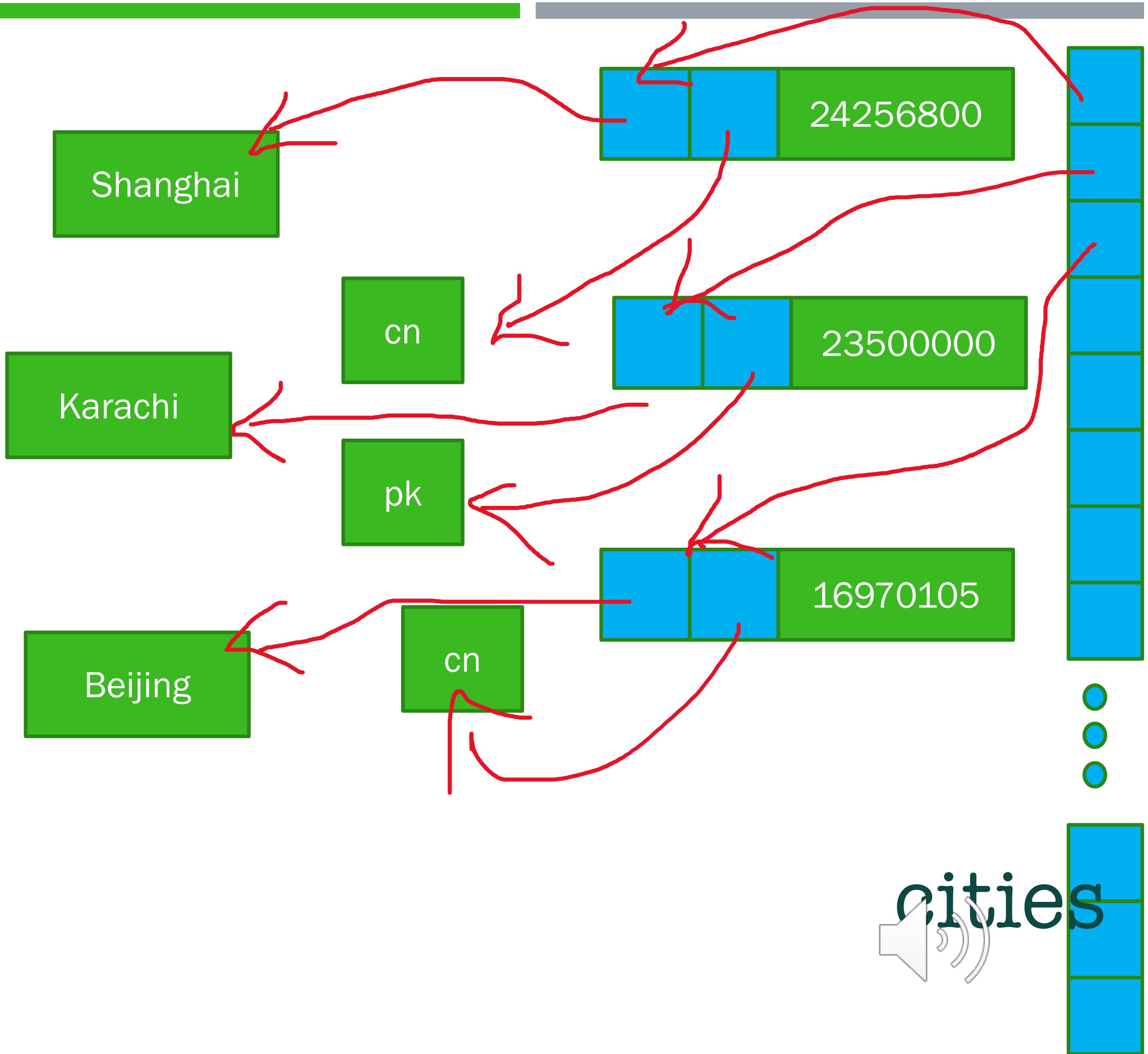


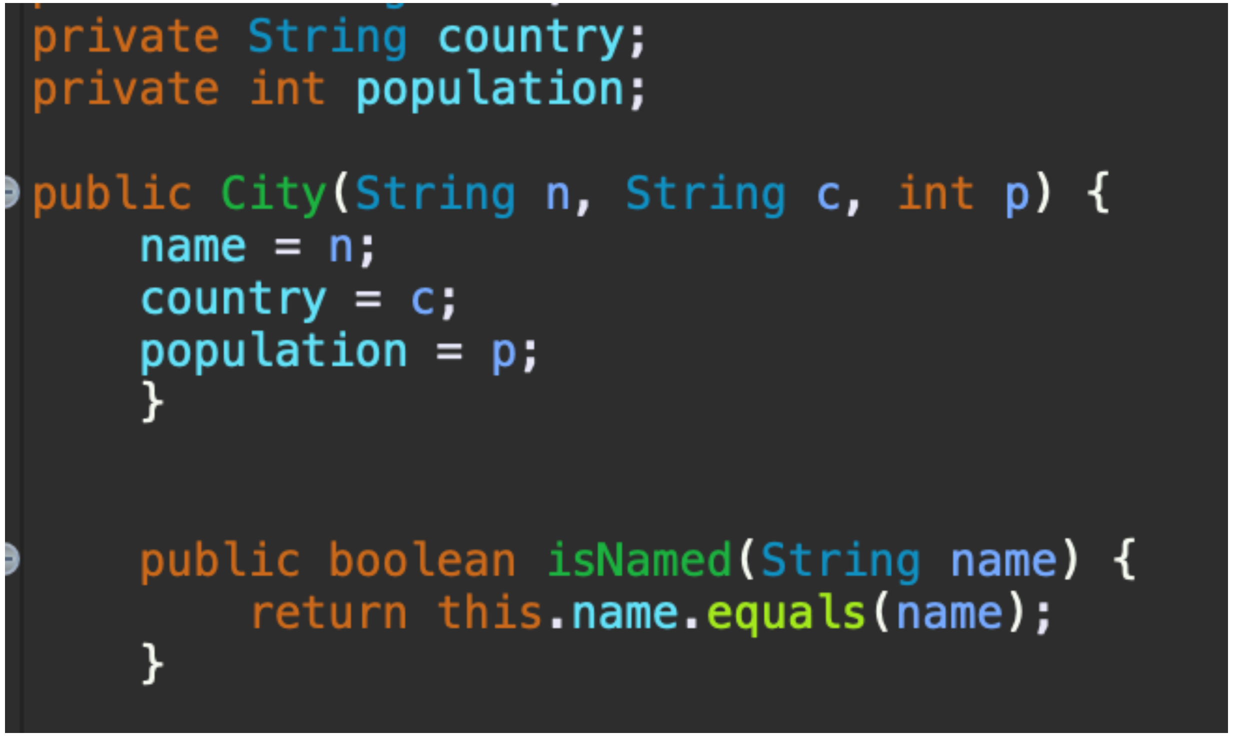
```
public static void main(String args[]) {
    City[] cities = new City[200];
```

Open the file to read it into the array

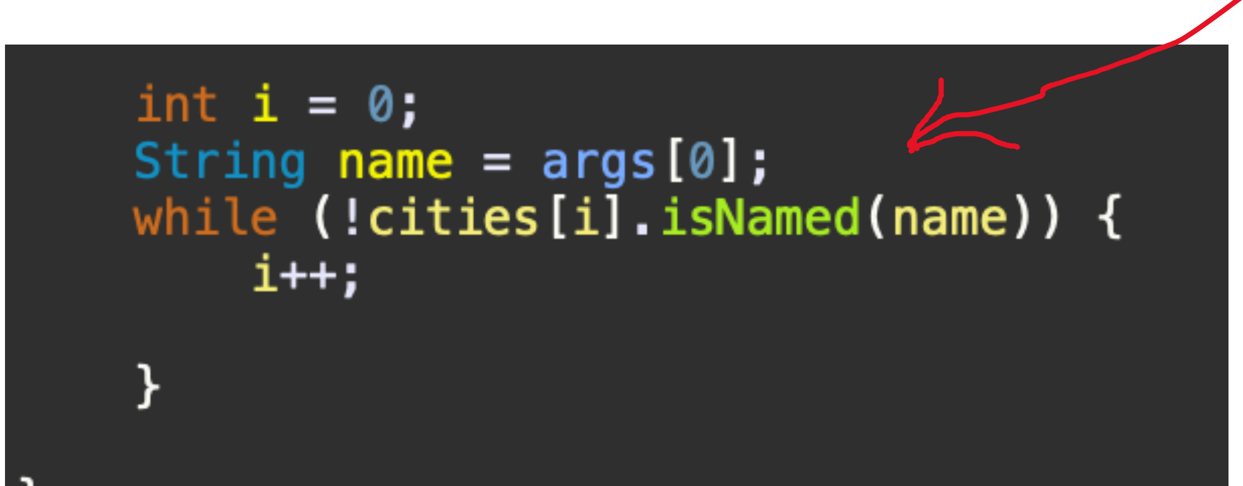
It's very common to load data in memory from external datafiles or from databases.

When we load everything in memory, every object (including Strings) becomes a reference to the actual values. To search the array, we'll just start at index 0 in the array and increase the index.





```
private String country;
private int population;

public City(String n, String c, int p) {
    name = n;
    country = c;
    population = p;
    }


    public boolean isNamed(String name) {
        return this.name.equals(name);
    }
```



```
    int i = 0;
    String name = args[0];
    while (!cities[i].isNamed(name)) {
        i++;

    }
}
```

Searching to see which city is named some particular string

We assumed that each name occurs once – means it is unique!

Time is proportional to the number of cities.

O(n)

Bigger = slower

We can do better with Arrays…

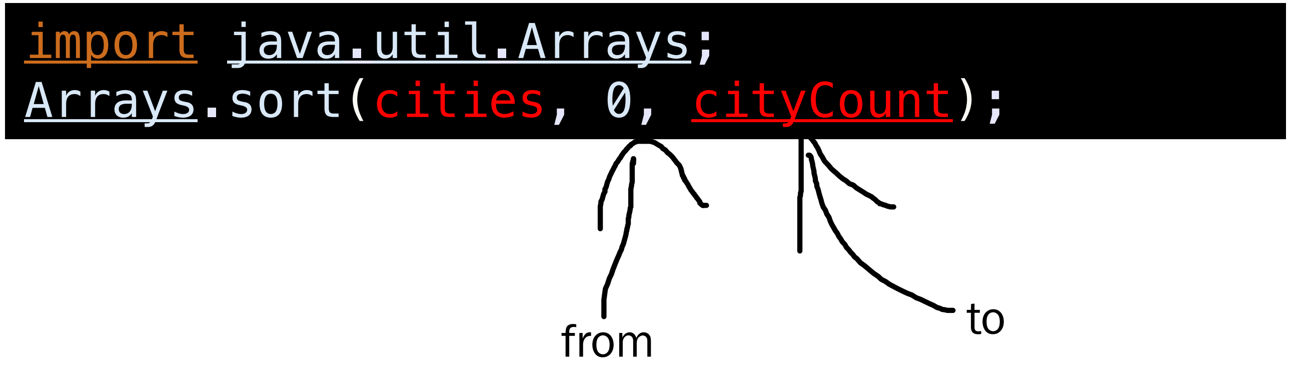
BINARY SEARCH

We can do far better with an array by running a binary search, which is the kind of search you run when looking for a word in a dictionary: you open in the middle, check a word there, and search either the first half or the second half.

I hope you recognized recursion intuitively…

You wouldn't be able to search a dictionary (otherwise than reading every page) if words were not ordered into it. A binary search can only work if the array is sorted. Class Arrays implements static methods that do that efficiently.

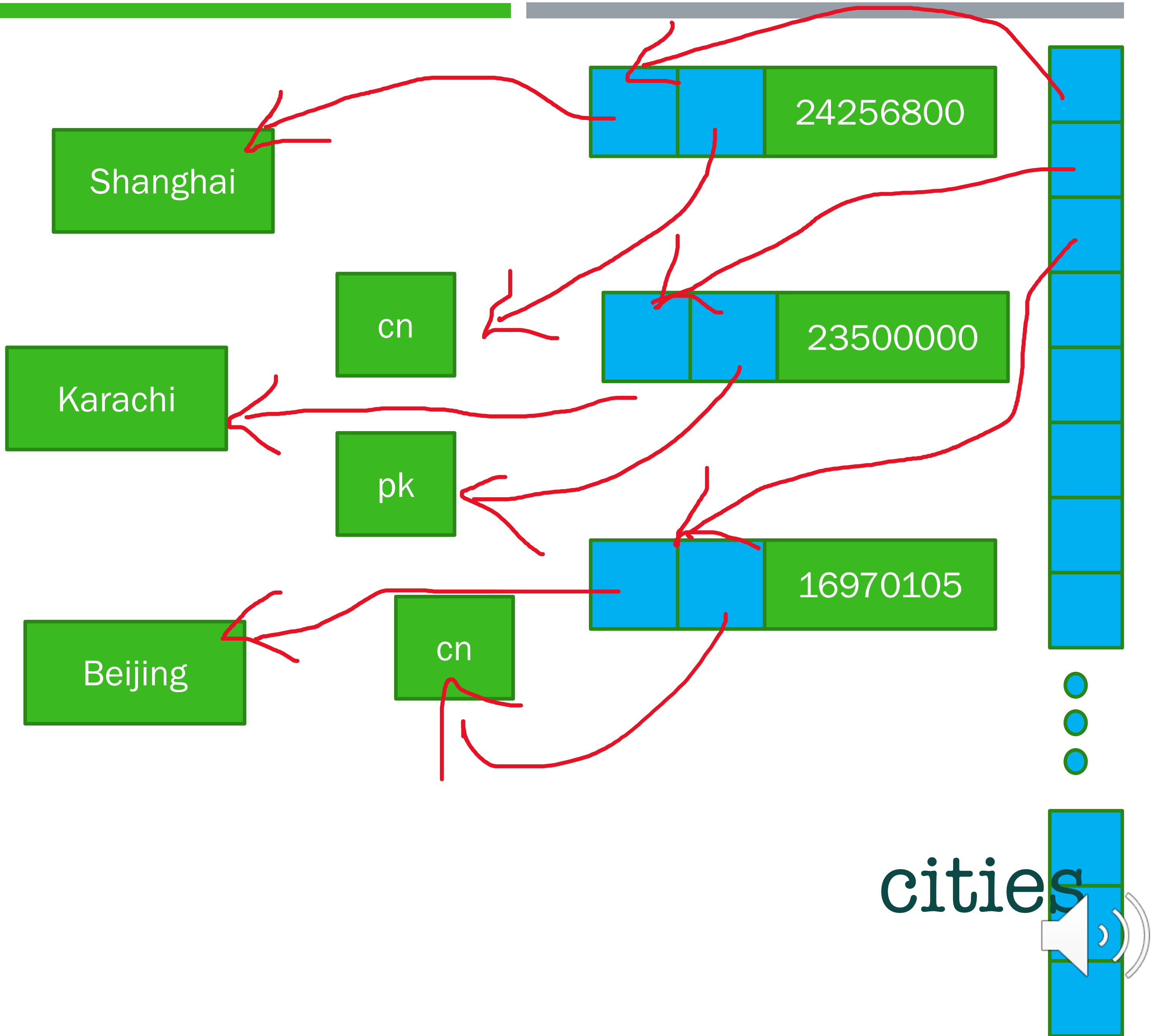
Precondition: the array must be sorted.



```
import java.util.Arrays;
Arrays.sort(cities, 0, cityCount);
```

Except objects can be complicated. What does “bigger” mean here?

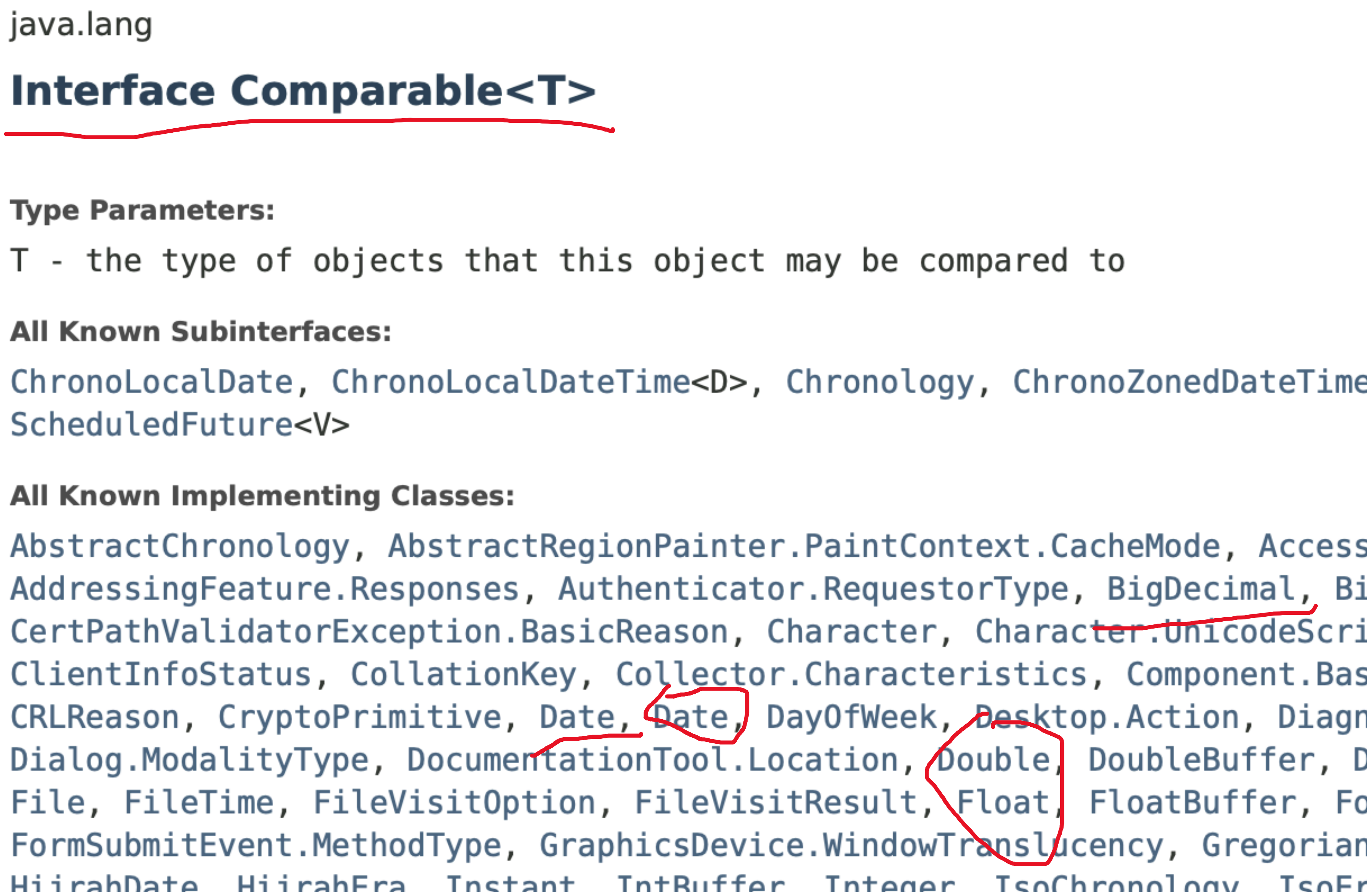
- Population?
- Lexographic order of names?
- Lexographic order of countries?
- Something else?



- Arrays.sort() needs to know...
- In fact, it will require the existence of a method called compareTo() that it will call for organizing objects in the correct order.

You can’t sort if you can’t

COMPARE



java.lang

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

All Known Subinterfaces:

ChronoLocalDate, ChronoLocalDateTime<D>, Chronology, ChronoZonedDateTime
ScheduledFuture<V>

All Known Implementing Classes:

AbstractChronology, AbstractRegionPainter.PaintContext.CacheMode, Access
AddressingFeature.Responses, Authenticator.RequestorType, BigDecimal, Bi
CertPathValidatorException.BasicReason, Character, Character.UnicodeScri
ClientInfoStatus, CollationKey, Collector.Characteristics, Component.Bas
CRLReason, CryptoPrimitive, Date, Date, DayOfWeek, Desktop.Action, Diagn
Dialog.ModalityType, DocumentationTool.Location, Double, DoubleBuffer, D
File, FileTime, FileVisitOption, FileVisitResult, Float, FloatBuffer, Fo
FormSubmitEvent.MethodType, GraphicsDevice.WindowTranslucency, Gregorian
HijrahDate, HijrahEra, Instant, IntBuffer, Integer, IsoChronology, IsoEr

REMINDER: INTERFACE

- When a specially named function has to exist in a class, it's said that the class must *implement* an interface. Let's review what an interface is.

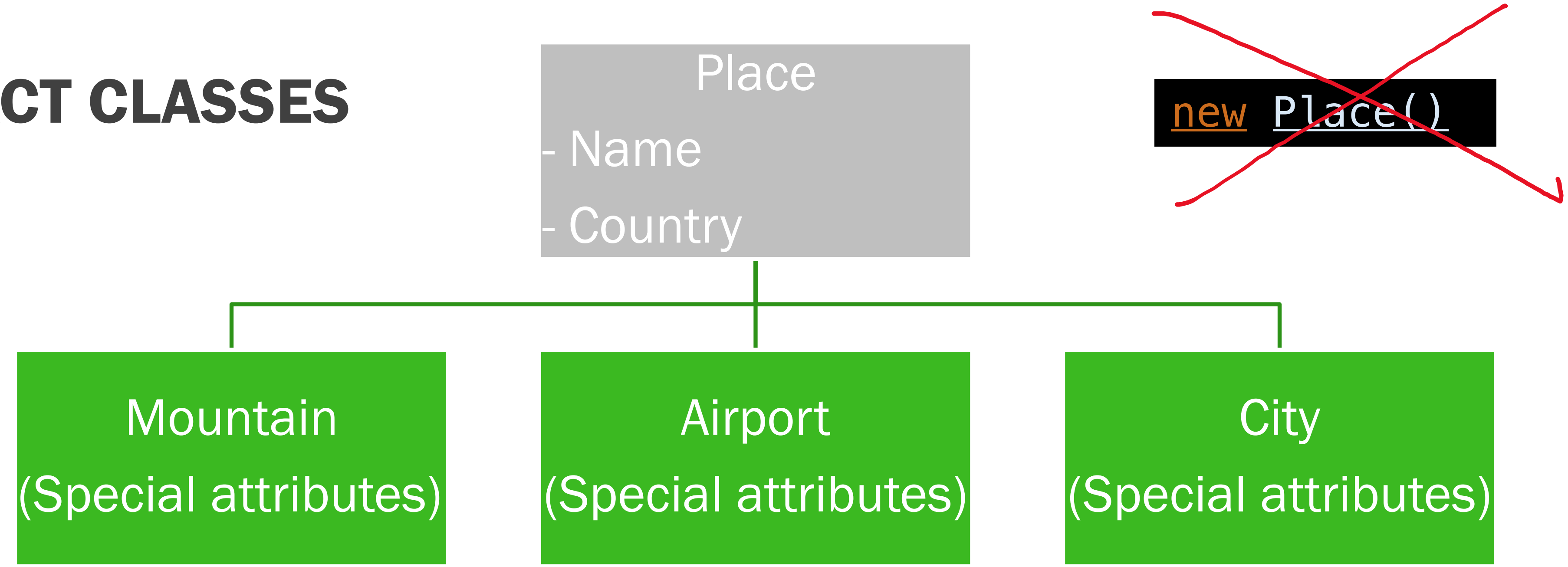
INTERFACES – LIGHTWEIGHT CLASSES (ALSO CAN CALL TEMPLATES)

- No variable attributes allowed
- Constants are allowed
- Define methods that classes must implement to conform

- Abstract class = implicit interface definition

Interfaces are special lightweight classes that mostly define a behavior, through methods. If a class says that it implements an interface, then it must have the methods defined in the interface.

ABSTRACT CLASSES

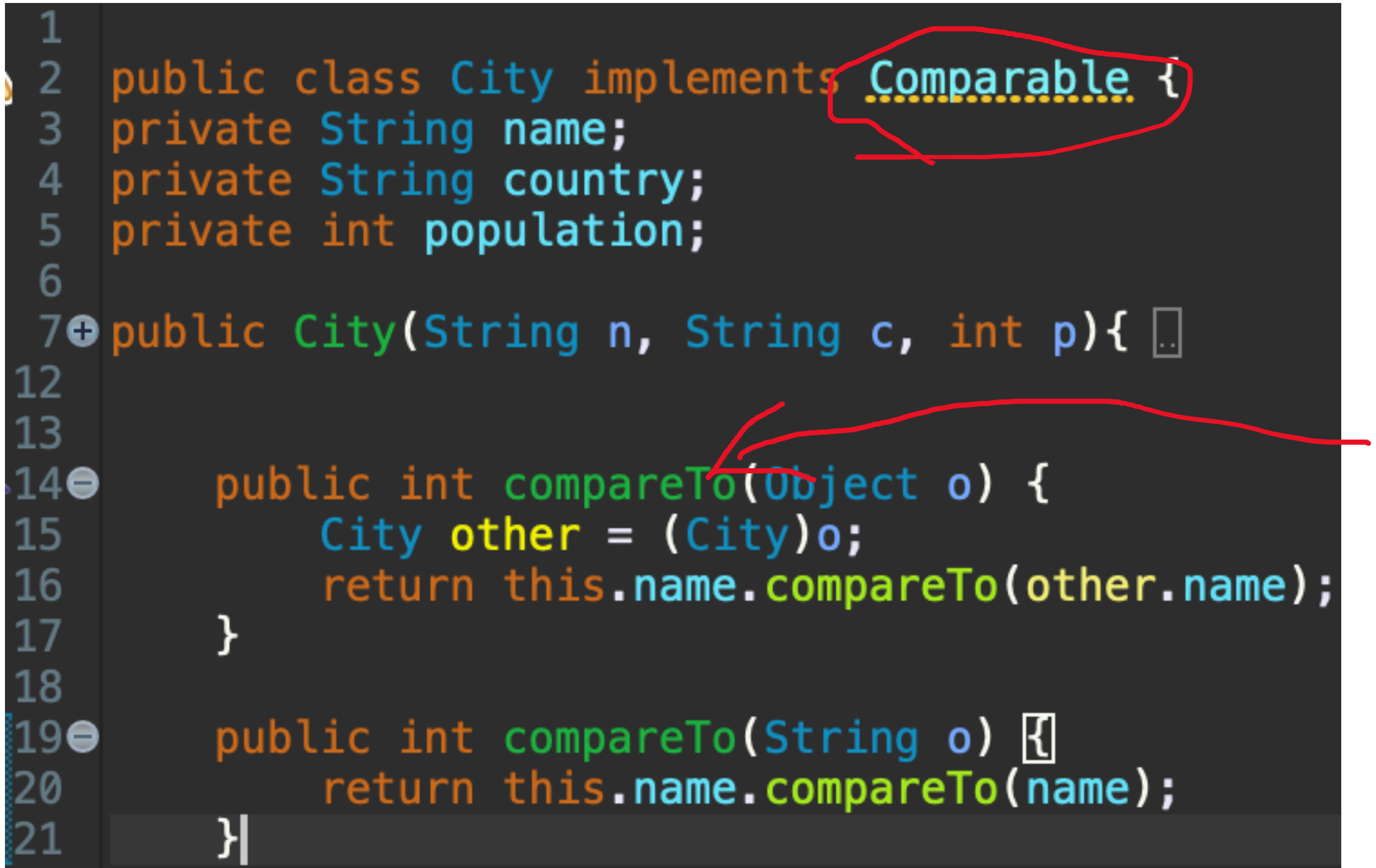


~~new Place()~~

- A class can also be declared Abstract (means the opposite of ”real”)
- An abstract class cannot be directly used to instantiate an object – instead you must first create a new class that extends the abstract one
- An abstract class can have methods defined, but also indicate that it **should** have a method and force others who extend the class to write the method in the child classes

Arrays.sort() only works if the class implements the Comparable interface, which requires a compareTo() method. Here we say that comparing cities means comparing their names.

Need for the Comparable interface.



```

public class City implements Comparable {
private String name;
private String country;
private int population;

public City(String n, String c, int p){


    public int compareTo(Object o) {
        City other = (City)o;
        return this.name.compareTo(other.name);
    }

    public int compareTo(String o) {
        return this.name.compareTo(name);
    }
```

Required by Arrays.sort()

Custom string comparator we will use

Find New York

0	Aberdeen
1	Boston
2	Chihuahua
3	Edinburgh
4	Istanbul
5	Liverppol
6	London
7	New York
8	Reykjavik
9	Rio De Janeiro
10	Shanghai
11	Tokyo

12 Elements

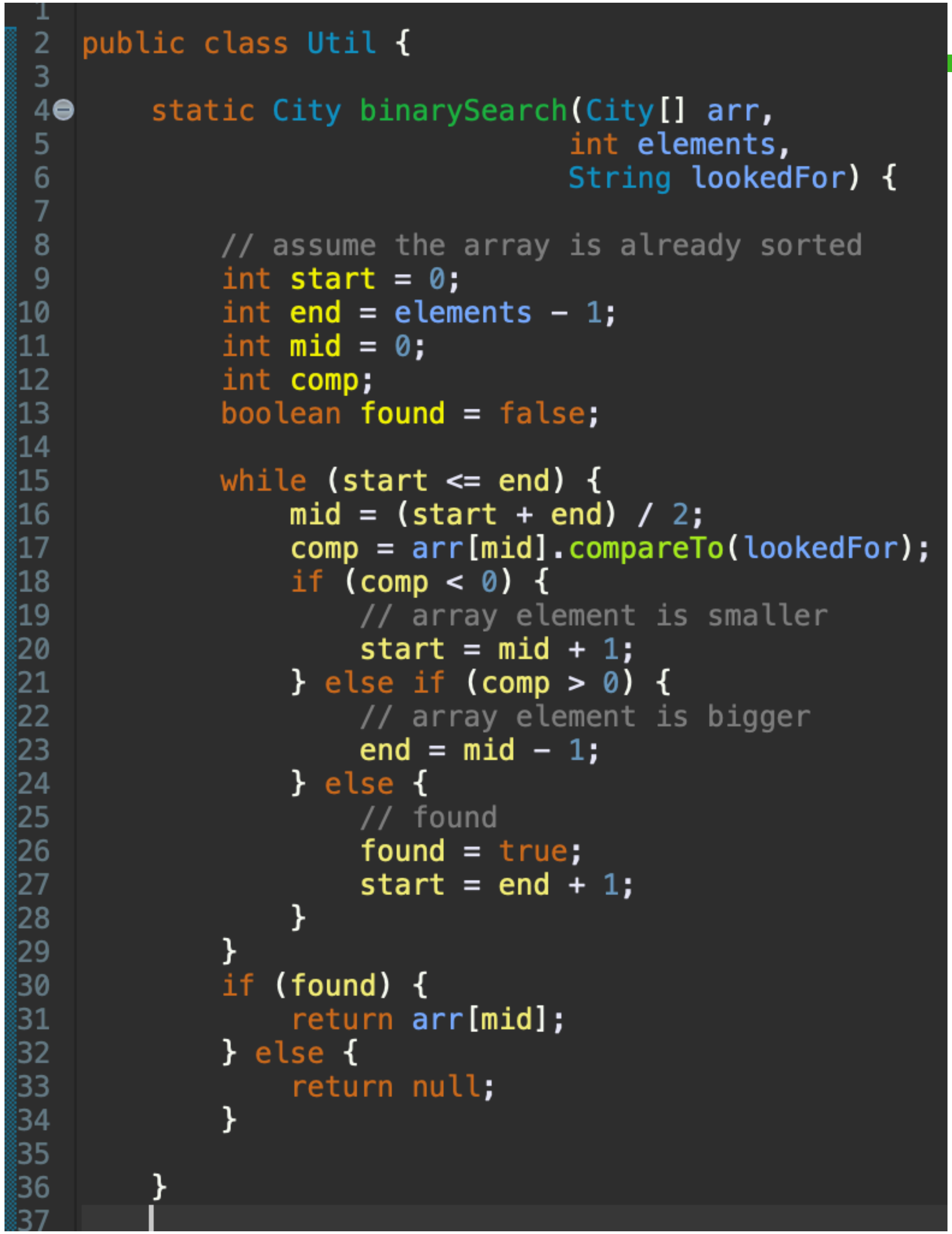
Middle = index 5
Search 6 to 11
New middle = index = 8
Search 6 to 7
New middle = index 6
Search 7 to 7
Found!

Binary search works by reducing the size of the part of the array that is searched at each step by half

NUMBER OF COMPARISONS IN BINARY SEARCH

- Size of the array is N
- Sequential search: N/2
  - If we double the number of items, we double the number of comparisons
- Binary Search: 2*log_2(N-1)
  - If we double the number of items we add one more comparison
  - It's a very efficient algorithm

$$O(\log n)$$



```java

public class Util {

    static City binarySearch(City[] arr,
                            int elements,
                            String lookedFor) {

        // assume the array is already sorted
        int start = 0;
        int end = elements - 1;
        int mid = 0;
        int comp;
        boolean found = false;

        while (start <= end) {
            mid = (start + end) / 2;
            comp = arr[mid].compareTo(lookedFor);
            if (comp < 0) {
                // array element is smaller
                start = mid + 1;
            } else if (comp > 0) {
                // array element is bigger
                end = mid - 1;
            } else {
                // found
                found = true;
                start = end + 1;
            }
        }
        if (found) {
            return arr[mid];
        } else {
            return null;
        }

    }
```

This is how it can be written in Java

Iteratively…

Or we can do it recursively

Trivial case? 0 or 1

Although this is a case where recursion doesn't make the code much simpler

Class Arrays contains several (static) binarySearch() methods. You don't need to write it ...

In practice these classic algorithms are part of Java's set of standard methods.