Computer Systems Design and Applications

CS209A

Lecture 4

Adam Ghandar

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.

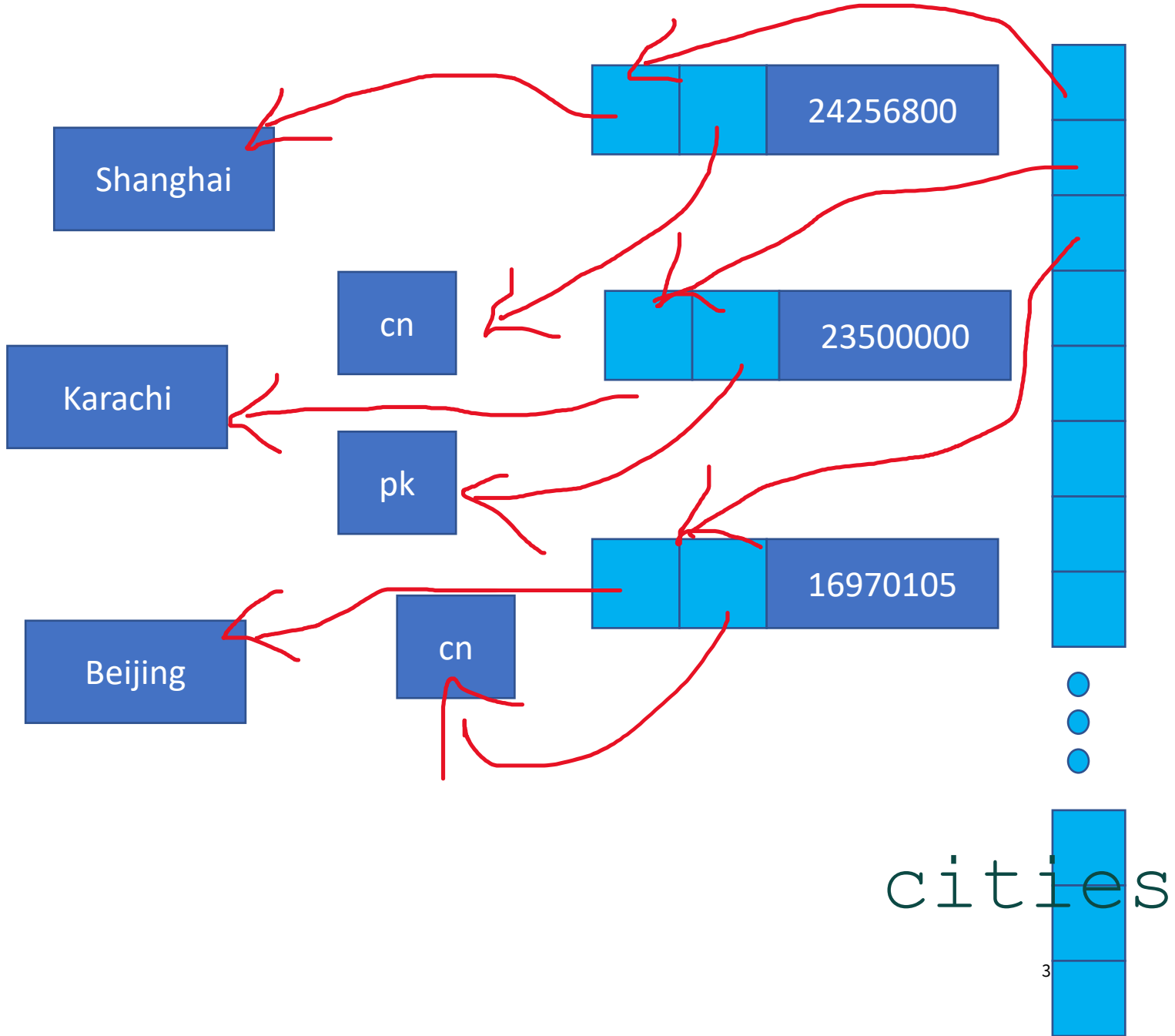
```
import java.util.Arrays;
Arrays.sort(cities, 0, cityCount);
```

from

to

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.

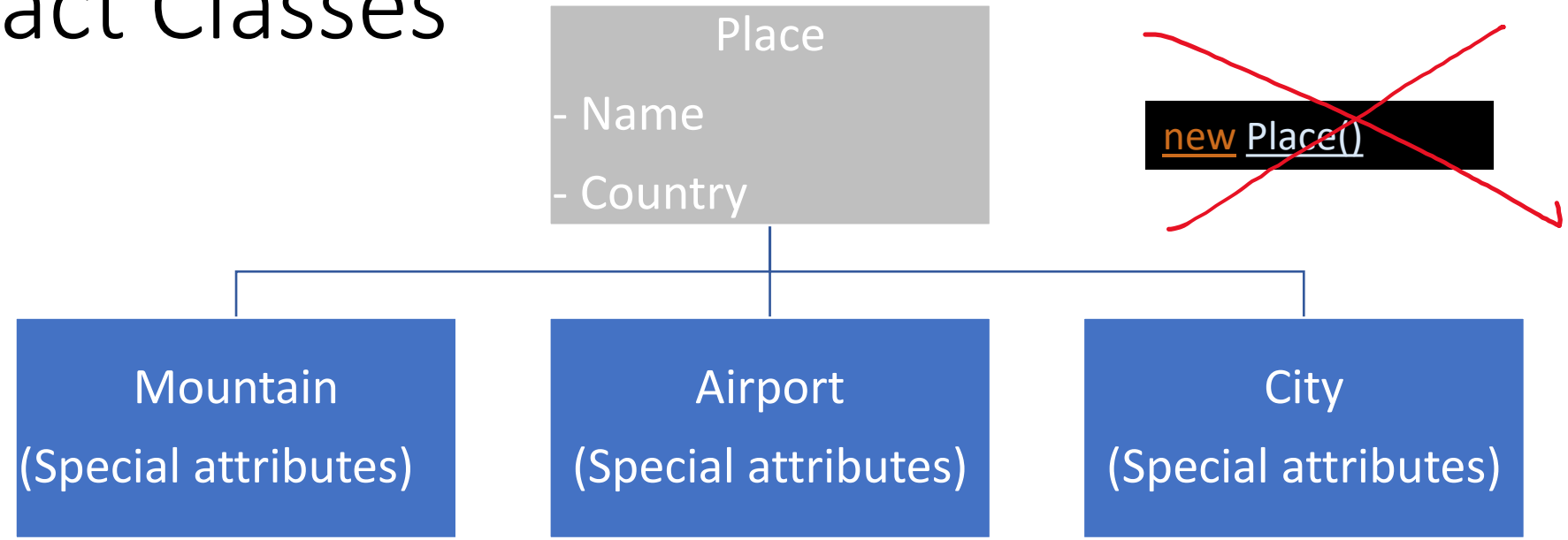
Interfaces – lightweight classes / templates

- 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



- 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.

```
1
2  public class City implements Comparable {
3  private String name;
4  private String country;
5  private int population;
6
7  public City(String n, String c, int p){ ..
12
13
14     public int compareTo(Object o) {
15         City other = (City)o;
16         return this.name.compareTo(other.name);
17     }
18
19     public int compareTo(String o) {
20         return this.name.compareTo(name);
21     }
```

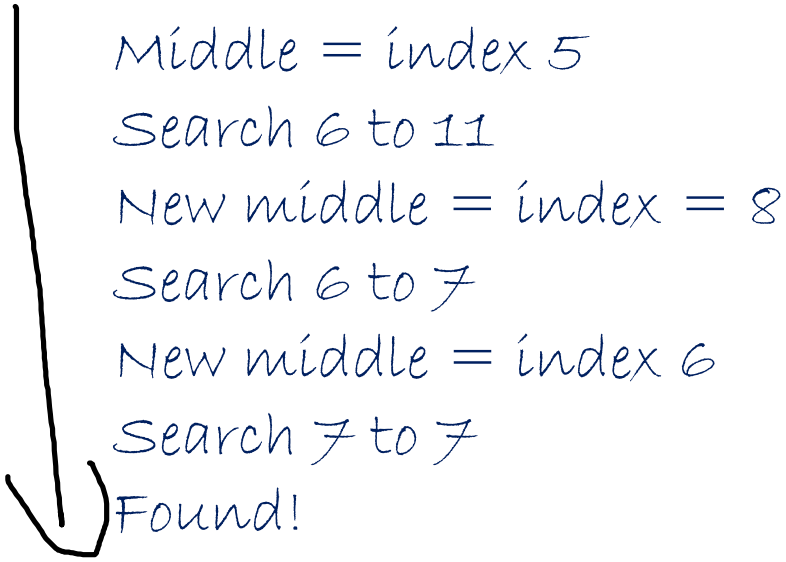
Required by Arrays.sort()

Custom string comparator we will use

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

Find New York

12 Elements



Remember we saw that 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 all that 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.

So…

What can we say about Arrays of Objects?

- Arrays of objects are good for some operations
- But for other operations not so much

What can we say about Arrays of Objects?

- They are not very good when data is dynamic
- Searching is efficient only when sorted (see previous presentation)
- Keeping order is hard if new objects are inserted randomly
  - When you keep adding and removing items in the array they are hard to manage (what to do with ”holes” in the array?). Efficient sorting only works when they are sorted – but if you want to insert values randomly and keep the order you have to move a lot of bytes around.

What can we say about Arrays of Objects?

The index isn’t always a natural way to access data

Additionally, the way you refer to an element in an array is its index (its position in the array). When you refer to a city, it's more natural to give the name of the city than to say "city at position n". In a program, that means that you would probably have the program user enter a name, search the array to find the position, then use the position afterwards.

List Data Type

- Using the methods we've just seen to search, access indexes and so on we have defined the basic operations we need for a list data type
- We could put them together in a class to specify a list data type
- Java has several data types for holding groups of objects including lists
- They are in the collections interface
- **ArrayList for example implements a list basically in the way I just described**

Java Collections Framework

- Classes and interfaces in java.util
- No predefined size
- A variety of container types that are suitable for different special grouping uses

You need to know what is important for your application

Java Collections defines interfaces and classes that allow you to group objects using different internal implementations. From simple arrays to more complex structures. There are different ways to store and organize objects, all with different features and levels of efficiency. Its good to understand this and choose efficient ones for particular problems. The relationship between structures for storing data and algorithms is also a big area of study.

An Overview of the Collections Framework

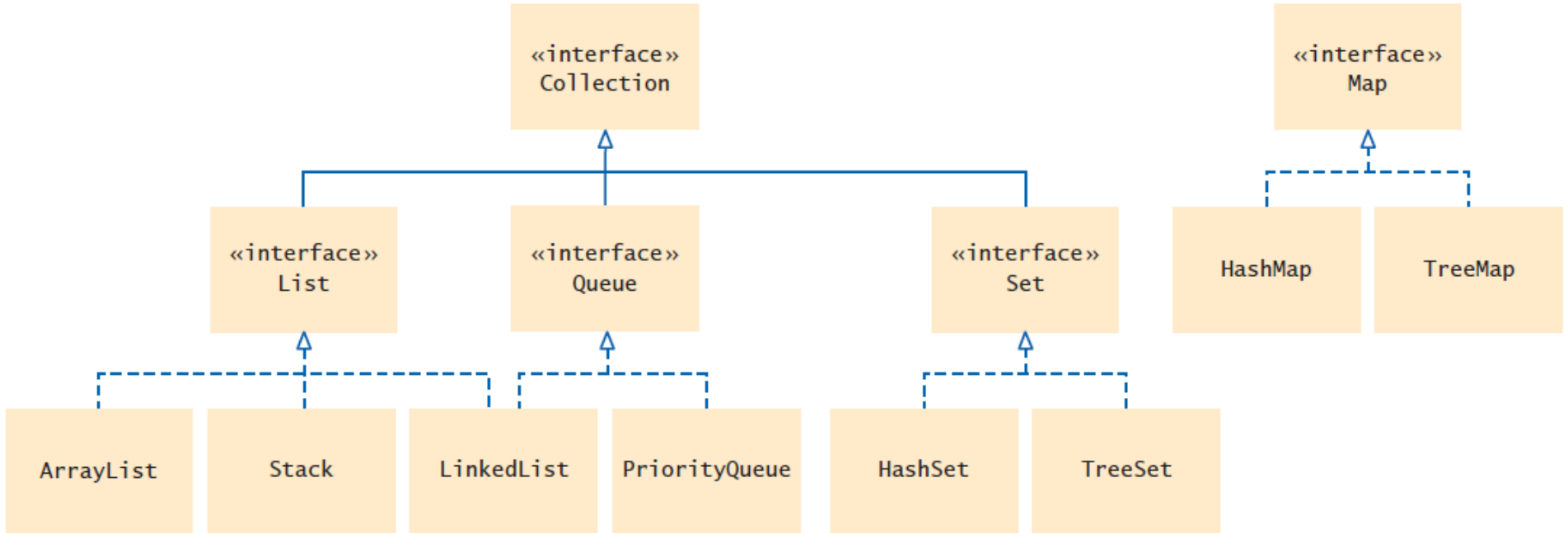
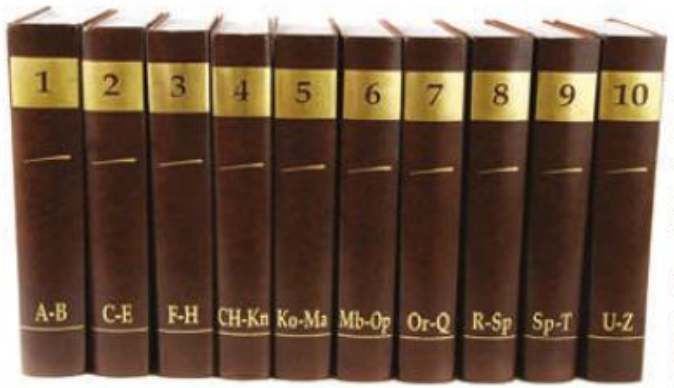


Figure 1 Interfaces and Classes in the Java Collections Framework



© Filip Fuxa/iStockphoto.

Figure 2 A List of Books


© parema/iStockphoto.

Figure 3 A Set of Books


© Vladimir Trenin/iStockphoto.

Figure 4 A Stack of Books

A list is a collection that remembers the order of its elements.

A stack is a collection of elements with "last-in, first-out" retrieval.

A set is an unordered collection of unique elements.

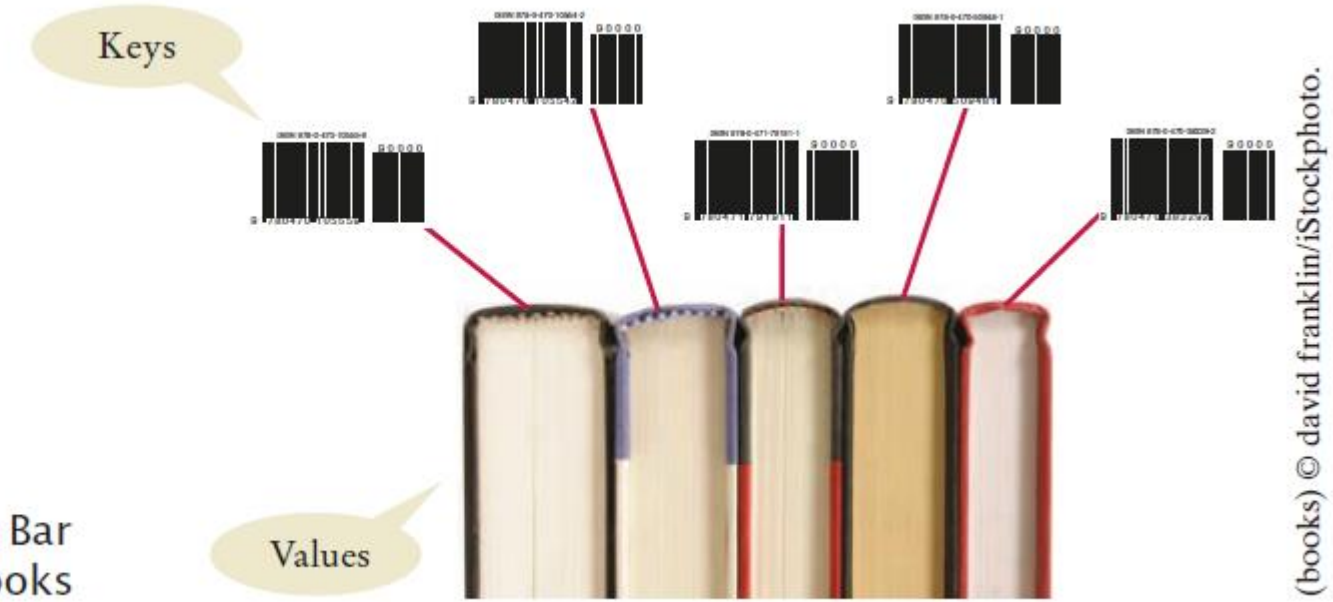


Figure 5
A Map from Bar Codes to Books

A map keeps associations between key and value objects.

Test

- A gradebook application stores a collection of quizzes. Should it use a list or a set?

Answer: A list is a better choice because the application will want to retain the order in which the quizzes were given.

- A student information system stores a collection of student records for a university. Should it use a list or a set?

Answer: A set is a better choice. There is no intrinsically useful ordering for the students. For example, the registrar's office has little use for a list of all students by their GPA. By storing them in a set, adding, removing, and finding students can be efficient.

- Why is a queue of books a better choice than a stack for organizing your required reading?

Answer: With a stack, you would always read the latest required reading, and you might never get to the oldest readings.

Table 1 The Methods of the `Collection` Interface

`Collection<String> coll = new ArrayList<>();`	The `ArrayList` class implements the `Collection` interface.
`coll = new TreeSet<>();`	The `TreeSet` class (Section 15.3) also implements the `Collection` interface.
`int n = coll.size();`	Gets the size of the collection. `n` is now `0`.
`coll.add("Harry");` `coll.add("Sally");`	Adds elements to the collection.
`String s = coll.toString();`	Returns a string with all elements in the collection. `s` is now `[Harry, Sally]`.
`System.out.println(coll);`	Invokes the `toString` method and prints `[Harry, Sally]`.
`coll.remove("Harry");` `boolean b = coll.remove("Tom");`	Removes an element from the collection, returning `false` if the element is not present. `b` is false.
`b = coll.contains("Sally");`	Checks whether this collection contains a given element. `b` is now `true`.

`for (String s : coll)`
`{`
`  System.out.println(s);`
`}`
	You can use the "for each" loop with any collection. This loop prints the elements on separate lines.

`Iterator<String> iter = coll.iterator();`	You use an iterator for visiting the elements in the collection (see Section 15.2.3).

An Overview of the Collections Framework

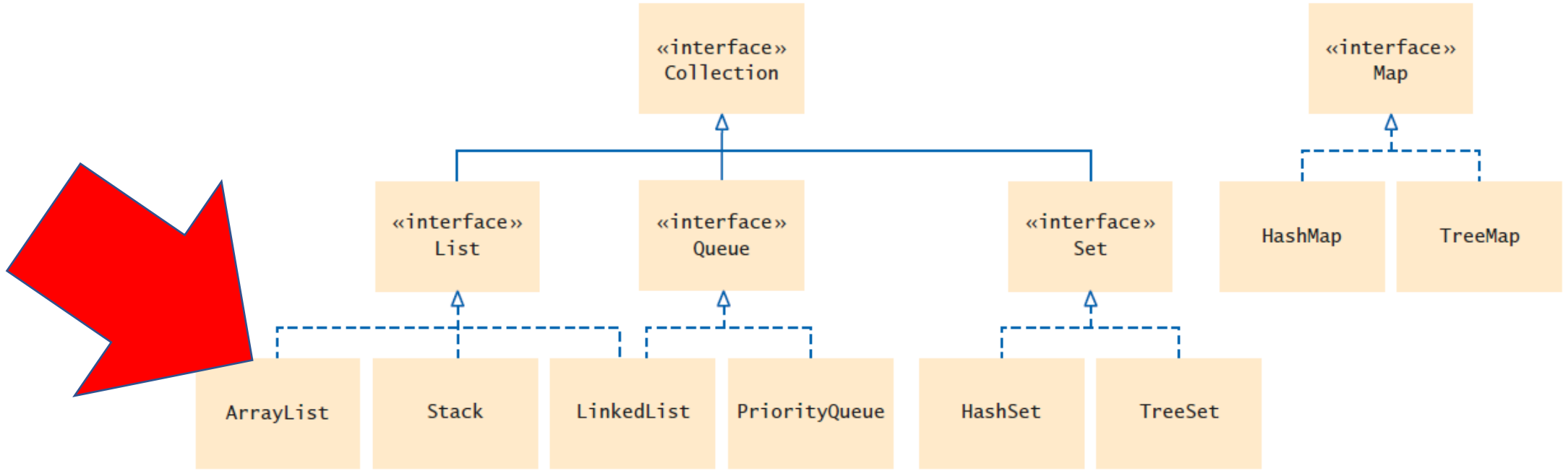


Figure 1 Interfaces and Classes in the Java Collections Framework

ArrayList is a Resizable Array

- There are different kinds of list and java.util.ArrayList is a dynamically resizable array

- `It grows automatically …`
- `… But it remains costly to resize`

  (Byte shifting still occurs)

The ArrayList that you have probably already used (if not, you may have used a Vector that is very much like an ArrayList) is such a collection. It grows automatically (which means that it automatically creates a bigger array when needed and copy the elements there – you need not do it yourself but it still happens)

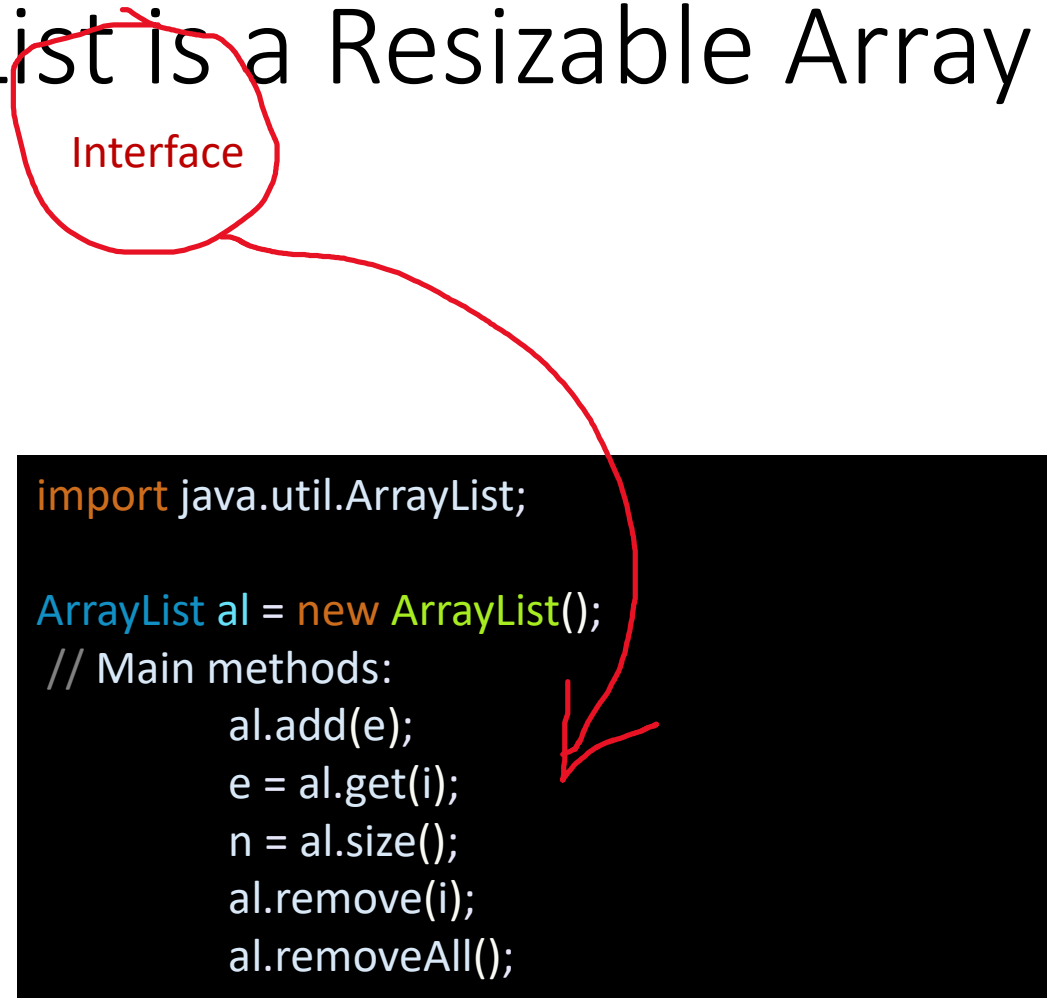
ArrayList is a Resizable Array

- You don’t use exactly the same syntax with an ArrayList as with an array, but you can basically do the same operations. Some methods apply to one element, others to all of them.

```
import java.util.ArrayList;

ArrayList al = new ArrayList();
 // Main methods:
                al.add(e);
                e = al.get(i);
                n = al.size();
                al.remove(i);
                al.removeAll();
```

careful as al[i] doesn't work!

ArrayList is a Resizable Array



"List" refers to the methods or operations that are available in the List ADT.

The array in the name refers to the implementation of the list using an array.

ArrayList is a Resizable Array

"Array" refers to how the data is actually physically stored.
You can offer the same operations but store data in different ways

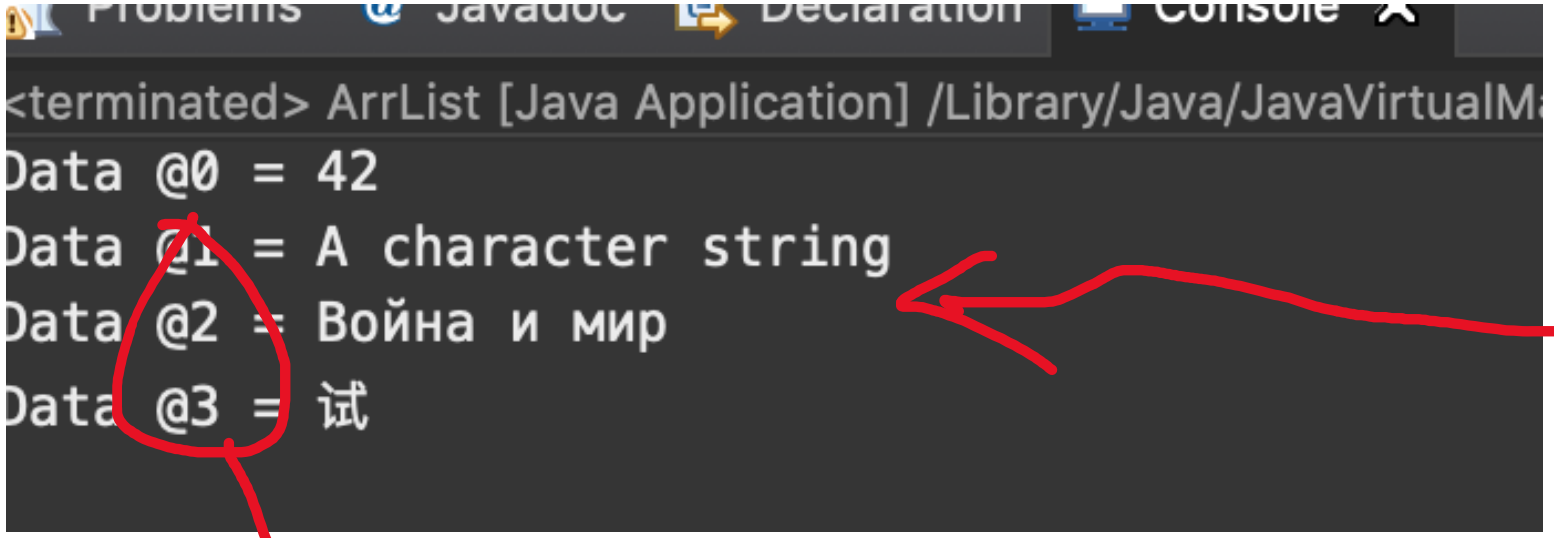
```
import java.util.ArrayList;

ArrayList al = new ArrayList();
 // Main methods:
                al.add(e);
                e = al.get(i);
                n = al.size();
                al.remove(i);
                al.removeAll();
```

```
public static void main(String [] args) {

    ArrayList al = new ArrayList();
    Random r = new Random();

    al.add(42);
    al.add("A character string");
    al.add("Война и мир");
    al.add(3.141592);
    al.add('试');

    int target = r.nextInt(al.size());
    al.remove(target);
    for (int i = 0; i < al.size(); i++) {
        System.out.println("Data @" + i + " = " + al.get(i));
    }
}
```

An ArrayList, like any collection, can store objects of any type, even of different types as here.

I'm removing a random element.

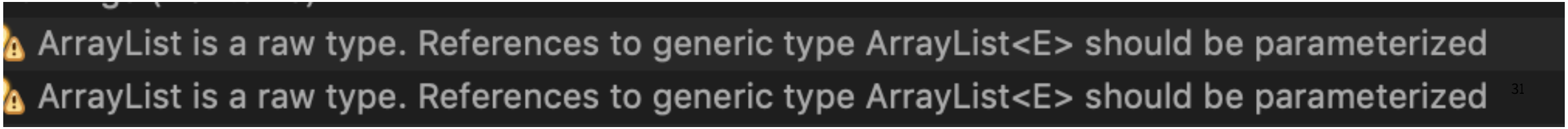
My random deletion removed the number pi. You see here that the array was rearranged and that everything is nicely displayed. But once again, a collection of different types of objects (you may have noticed autoboxing in action) is a bad idea. In a collection, the type of all items should be the same.



pi is gone

Indices are renumbered

Note that mixing very different objects is a poor programming practice, and the javac compiler isn't too happy with it. You can still run the program though.



Collections have been typed...

```
ArrayList<Object> al = new ArrayList<Object>();
```

And as you have seen in the previous example that may be typed with a generic type.

```
MyGenericList<String> nameList;
MyGenericList<Float> distanceList;
```

Collections are definitely where you want to use generics.

GENERICS

Defining templates for javac

```java
public class ArrList {

    public static void main(String [] args) {

        ArrayList<Object> al = new ArrayList<Object>();

        Random r = new Random();

        al.add(42);
        al.add("A character string");
        al.add("Война и мир");
        al.add(3.141592);
        al.add('试');

        int target = r.nextInt(al.size());
        al.remove(target);
        for (int i = 0; i < al.size(); i++) {
            System.out.println("Data @" + i + " = " + al.get(i));
        }
    }
```

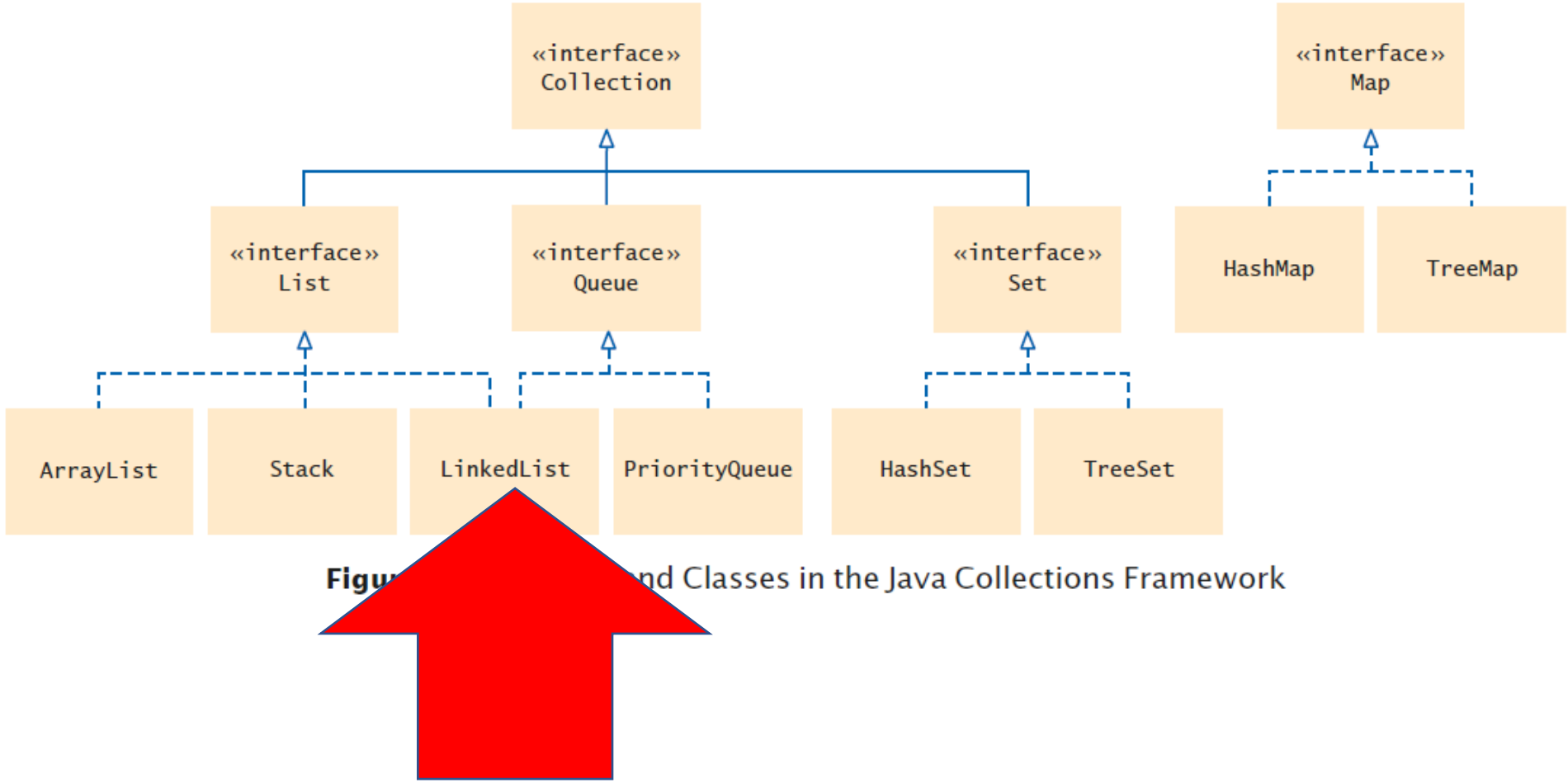
Can use the fact all items are objects to hold different object types in a string (works here as all the types have a meaningful “toString()” method implemented.

This time I deleted the integer 42

```
<terminated> ArrList [Java Application] /Library/Java
Data @0 = A character string
Data @1 = Война и мир
Data @2 = 3.141592
Data @3 = 试
```

```java
import java.util.ArrayList;

public class MyGenericList <T>{

    private ArrayList<T> _list;

    public void setElement(T o) {
        _list.add(o);
    }

    public T getElement(int i) {
        return _list.get(i);
    }
}
```

If you need to have your special wrapper around a collection, you use a generic type (here T) that you pass to the collection.

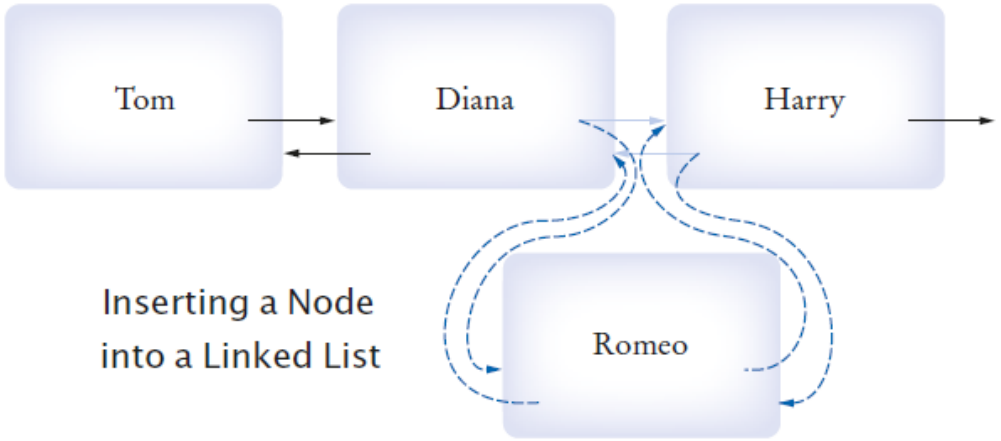


Figu nd Classes in the Java Collections Framework

Linked Lists

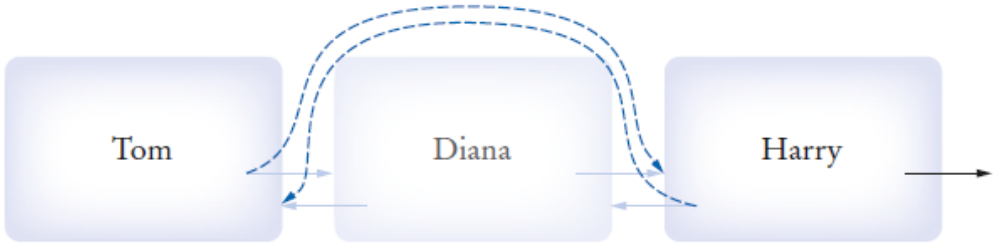
A linked list consists of a number of nodes, each of which has a reference to the next node.

Adding and removing elements at a given location in a linked list is efficient.



Inserting a Node into a Linked List

Visiting the elements of a linked list in sequential order is efficient, but random access is not.



Removing a Node from a Linked List

Linked Lists

Use linked lists when the efficiency of inserting or removing elements is important (avoid leaving “holes”) and you rarely need element access in random order.

As we saw use Generic class Specify type of elements in angle brackets: eg
LinkedList<Product>Package

Table 2 Working with Linked Lists

`LinkedList<String> list = new LinkedList<>();`	An empty list.
`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements (see Table 3 on page 698).

List Iterator

Initially points before the first element.
Move the position with next method:

```
if (iterator.hasNext()) {
iterator.next();
}
```

The next method returns the element that the iterator is passing.

The return type of the next method matches the list iterator's type parameter.

List Iterator

```
while (iterator.hasNext()) {
   String name = iterator.next();
  //Do something with name
}
// To use the “for each” loop:
for (String name : employeeNames) {
  //Do something with name
}
```

List Iterator The nodes of the LinkedList class store two links: One to the next element One to the previous one Called a doubly-linked list
To move the list position backwards,

```
while (iterator.hasPrevious()) {
   String name = iterator.previous();
  //Do something with name
}
```

The iterator can also be used to add and remove elements eg iterator.add(element)

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator();
```

Table 3 Methods of the Iterator and ListIterator Interfaces

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.
`iter.previous();` `iter.set("Juliet");`	The `set` method updates the last element returned by `next` or `previous`. The list is now `[Juliet]`.
`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`  s = iter.previous();`
`}`
	`hasPrevious` returns `true` because the iterator is not at the beginning of the list. `previous` and `hasPrevious` are `ListIterator` methods.

`iter.add("Diana");`	Adds an element before the iterator position (`ListIterator` only). The list is now `[Diana, Juliet]`.
`iter.next();` `iter.remove();`	`remove` removes the last element returned by `next` or `previous`. The list is now `[Diana]`.

Use a list iterator to access elements inside a linked list.
Encapsulates a position anywhere inside the linked list.
Think of an iterator as pointing between two elements:
Analogy: like the cursor in a word processor points between two characters

An Overview of the Collections Framework

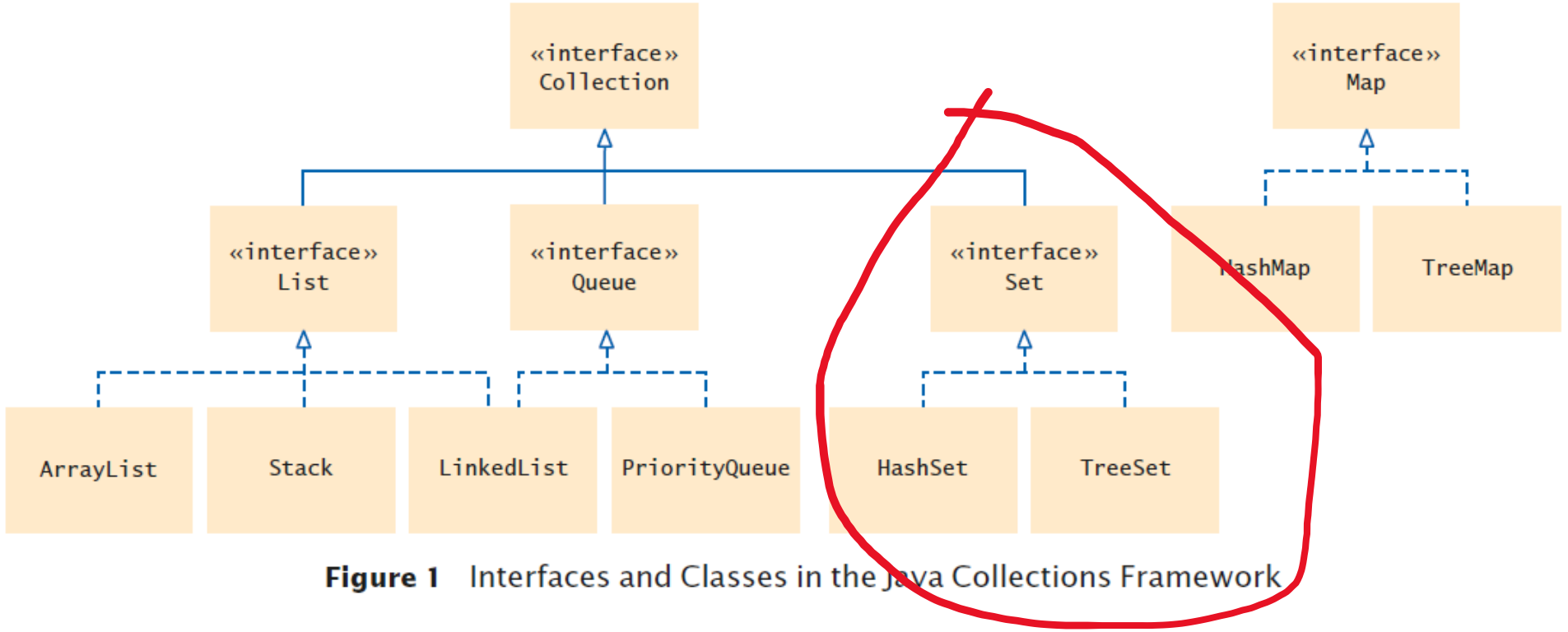


Figure 1 Interfaces and Classes in the Java Collections Framework

The `HashSet` and `TreeSet` classes both implement the `Set` interface.

Set implementations arrange the elements so that they can locate them quickly.

You can form hash sets holding objects of type `String`, `Integer`, `Double`, `Point`, `Rectangle`, or `Color`.

You can form tree sets for any class that implements the `Comparable` interface, such as `String` or `Integer`.

When you construct a HashSet or TreeSet, store the reference in a Set variable.

```
Set<String> names = new HashSet<>();
```

or

```
Set<String> names = new TreeSet<>();
```

Tree set or Hashset?

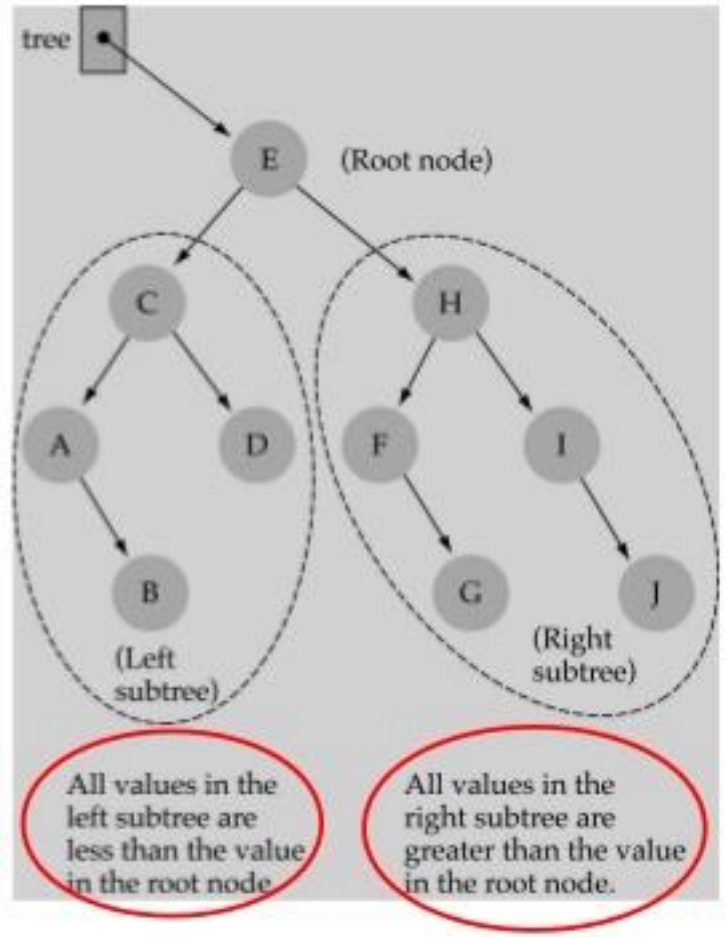
Use a TreeSet if you want to visit the set's elements in sorted order. Otherwise choose a HashSet.It is a bit more efficient.

A good hash function (certainly different) hash values for each object so that they are scattered about in a hash table.

Binary Search Trees

- **Binary Search Tree Property**:

  The value stored at a node is *greater* than the value stored at its left child and *less* than the value stored at its right child



Set<String> names
= new HashSet<>();

Table 4 Working with Sets

`Set<String> names;`	Use the interface type for variable declarations.
`names = new HashSet<>();`	Use a `TreeSet` if you need to visit the elements in sorted order.
`names.add("Romeo");`	Now `names.size()` is 1.
`names.add("Fred");`	Now `names.size()` is 2.
`names.add("Romeo");`	`names.size()` is still 2. You can't add duplicates.
`if (names.contains("Fred"))`	The `contains` method checks whether a value is contained in the set. In this case, the method returns `true`.
`System.out.println(names);`	Prints the set in the format `[Fred, Romeo]`. The elements need not be shown in the order in which they were inserted.
`for (String name : names) { . . . }`	Use this loop to visit all elements of a set.
`names.remove("Romeo");`	Now `names.size()` is 1.
`names.remove("Juliet");`	It is not an error to remove an element that is not present. The method call has no effect.

Iterator<String> iter = names.iterator(); while (iter.hasNext())

An Overview of the Collections Framework

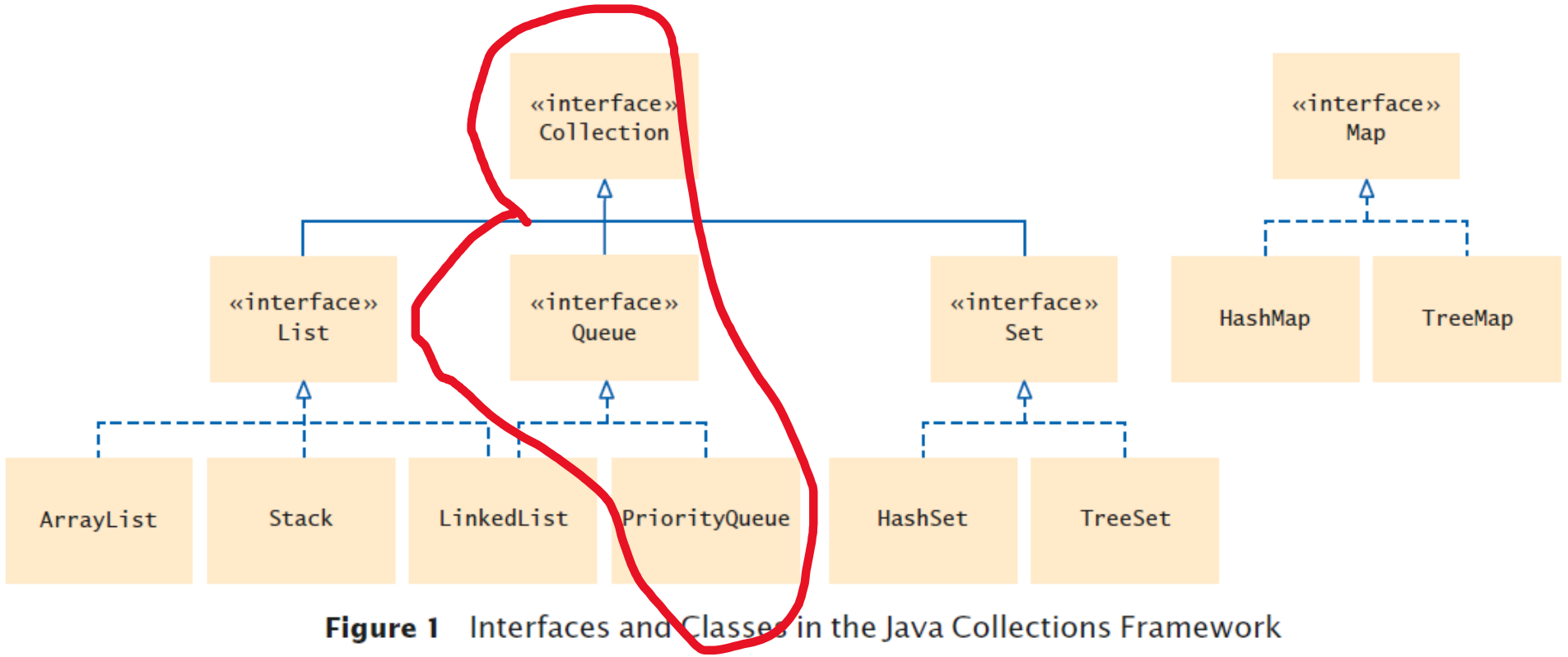


Figure 1 Interfaces and Classes in the Java Collections Framework

Table 8 Working with Queues

`Queue<Integer> q = new LinkedList<>();`	The `LinkedList` class implements the `Queue` interface.

`q.add(1);`
`q.add(2);`
`q.add(3);`
	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.

`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.

The LinkedList class implements the Queue interface. When you need a queue, initialize a Queue variable with a LinkedList object:

```
Queue<String> q = new LinkedList<>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0) {
System.out.print(q.remove() + " ");
}
```

Table 9 Working with Priority Queues

`PriorityQueue<Integer> q = new PriorityQueue<>();`	This priority queue holds `Integer` objects. In practice, you would use objects that describe tasks.
`q.add(3); q.add(1); q.add(2);`	Adds values to the priority queue.
`int first = q.remove();` `int second = q.remove();`	Each call to `remove` removes the most urgent item: `first` is set to 1, `second` to 2.
`int next = q.peek();`	Gets the smallest value in the priority queue without removing it.

Because the priority queue needs to be able to tell which element is the smallest, the added elements should belong to a class that implements the `Comparable` interface. Thus, each removal operation extracts the *minimum* element from the queue.

An Overview of the Collections Framework

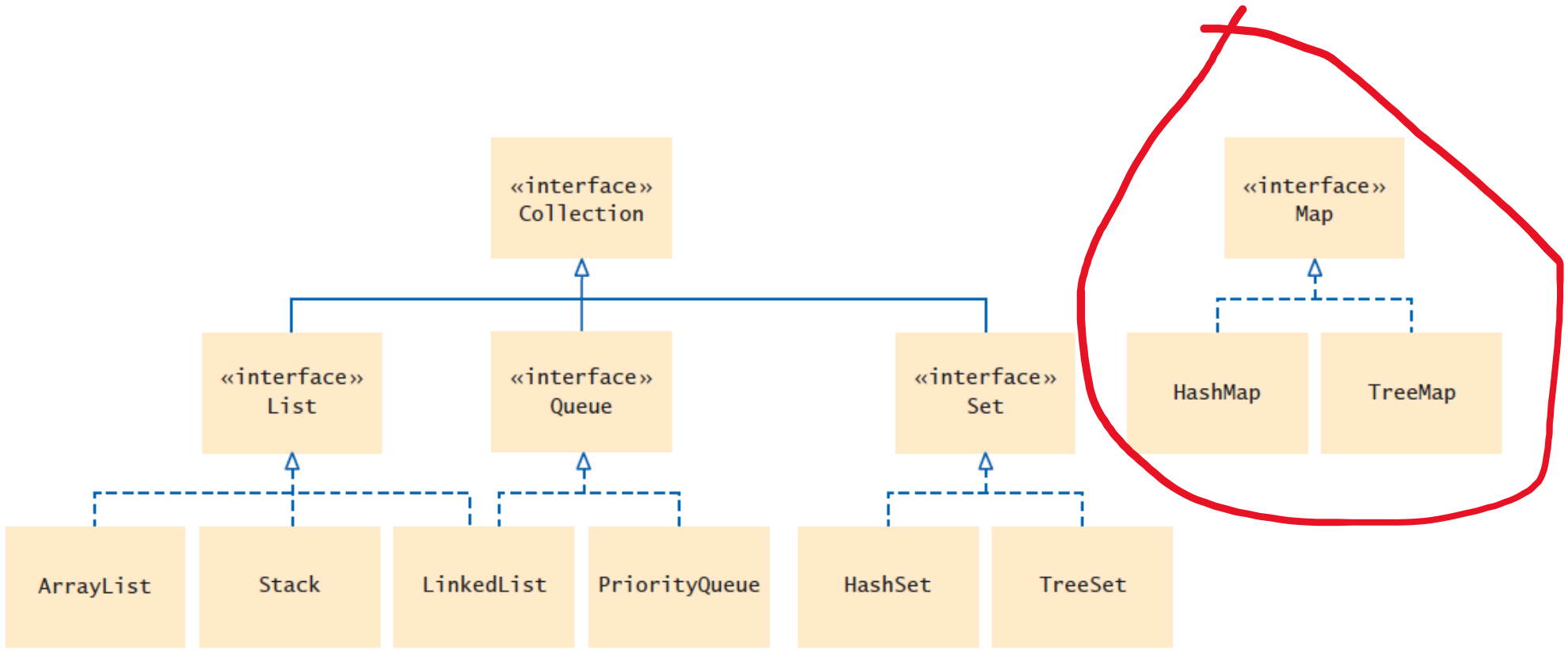
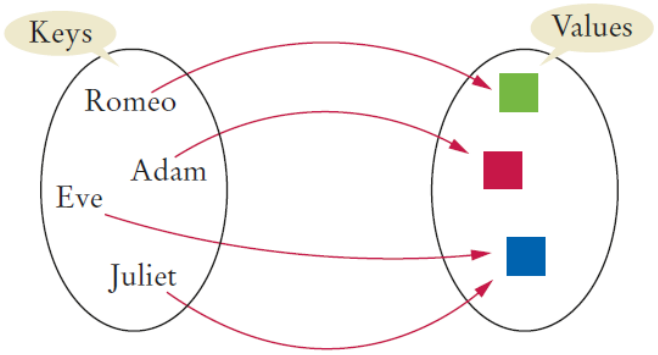


Figure 1 Interfaces and Classes in the Java Collections Framework

Maps

The HashMap and TreeMap classes both implement the Map interface.



A Map

In a *TreeMap*, the key/value associations are stored in a sorted tree, in which they are sorted according to their `keys`. For this to work, it must be possible to compare the keys to one another.

This means either that the keys must implement the interface *Comparable<K>*, or that a *Comparator* must be provided for comparing keys.
(The *Comparator* can be provided as a parameter to the *TreeMap* constructor.)

Note that in a *TreeMap*, as in a *TreeSet*, the `compareTo()` (or `compare()`) method is used to decide whether two keys are to be considered the same.

After constructing a HashMap or TreeMap, you can store the reference to the map object in a Map reference:

```
Map<String, Color> favoriteColors = new HashMap<>();
```

Map<String, Color> favoriteColors = new HashMap<>();

Suppose that `map` is a variable of type *Map<K,V>* for some specific types *K* and *V*.
Then the following are some of the methods that are defined for `map`:

- `map.get(key)` — returns the object of type *V* that is associated by the map to the `key`. If the map does not associate any value with `key`, then the return value is `null`. Note that it's also possible for the return value to be `null` when the map explicitly associates the value `null` with the key. Referring to "`map.get(key)`" is similar to referring to "`A[key]`" for an array `A`. (But note that there is nothing like an *IndexOutOfBoundsException* for maps.)

Color julietsFavoriteColor =
favoriteColors.get("Juliet");

- `map.put(key,value)` — Associates the specified `value` with the specified `key`, where `key` must be of type *K* and `value` must be of type *V*. If the map already associated some other value with the key, then the new value replaces the old one. This is similar to the command "`A[key] = value`" for an array.

favoriteColors.put("Juliet", Color.BLUE);

- `map.putAll(map2)` — if `map2` is another map of type *Map<K,V>*, this copies all the associations from `map2` into map.
- `map.remove(key)` — if `map` associates a value to the specified `key`, that association is removed from the map.

favoriteColors.remove("Juliet");

- `map.containsKey(key)` — returns a boolean value that is `true` if the map associates some value to the specified `key`.
- `map.containsValue(value)` — returns a boolean value that is `true` if the map associates the specified `value` to some key.

Maps

Table 5 Working with Maps	
`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<>();`	Use a `HashMap` if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.
`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

`for (String key : scores.keySet()) {`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

If `map` is a variable of type *Map<K,V>*, then

The value returned by `map.keySet()` is a "view" of keys in the map implements the *Set<K>* interface

```
Set<String> keySet = m.keySet();
for (String key : keySet){
Color value = m.get(key);
System.out.println(key + "->" + value);
}
```

`map.values()` returns an object of type *Collection<V>* that contains all the values from the associations that are stored in the map. The return value is a *Collection* rather than a *Set* because it can contain duplicate elements.

One of the things that you can do with a *Set* is get an *Iterator* for it and use the iterator to visit each of the elements of the set in turn.

`map.entrySet()` returns a set that contains all the associations from the map.

The elements in the set are objects of type *Map.Entry<K,V>*.

The return type is written as *Set<Map.Entry<K,V>>*.

Each `Map.Entry` object contains one key/value pair, and defines methods `getKey()` and `getValue()` for retrieving the key and the value.

Pop quiz

- What is the difference between a set and a map?

Answer: A set stores elements. A map stores associations between keys and values.

- Why is the collection of the keys of a map a set and not a list?

Answer: The ordering does not matter, and you cannot have duplicates

- Why is the collection of the values of a map not a set?

Answer: Because it might have duplicates.

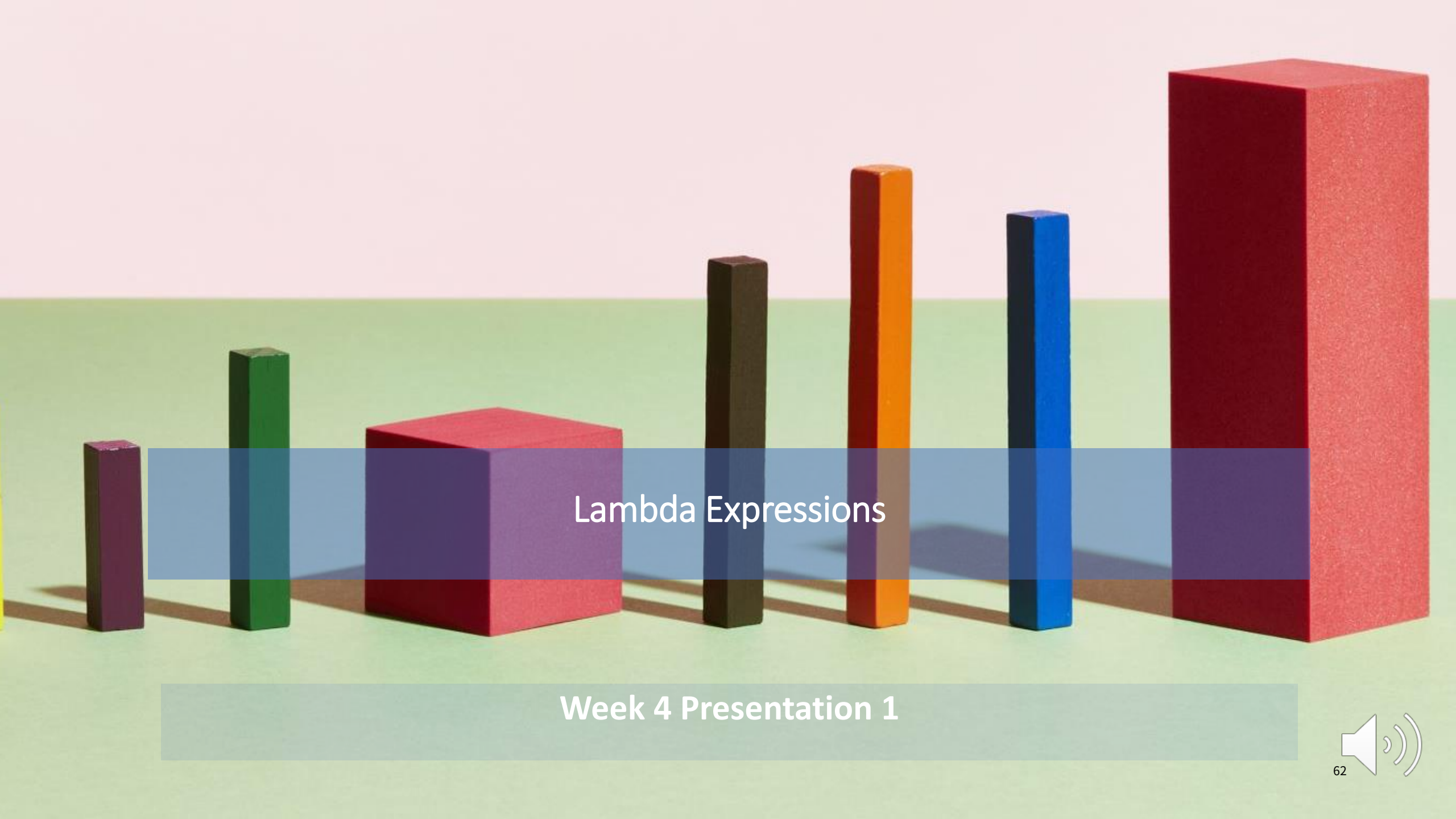
Table 7 Working with Stacks

`Stack<Integer> s = new Stack<>();`	Constructs an empty stack.

`s.push(1);`
`s.push(2);`
`s.push(3);`
	Adds to the top of the stack; `s` is now `[1, 2, 3]`. (Following the `toString` method of the `Stack` class, we show the top of the stack at the end.)

`int top = s.pop();`	Removes the top of the stack; `top` is set to 3 and `s` is now `[1, 2]`.
`head = s.peek();`	Gets the top of the stack without removing it; `head` is set to 2.

```
if (!s.empty()) …
if (s.size() > 0) …
```



Lambda Expressions

Week 4 Presentation 1

Lambda is a letter in the Greek alphabet that was used by the mathematician Alonzo Church in his study of computable functions. His lambda notation makes it possible to define a function without giving it a name.

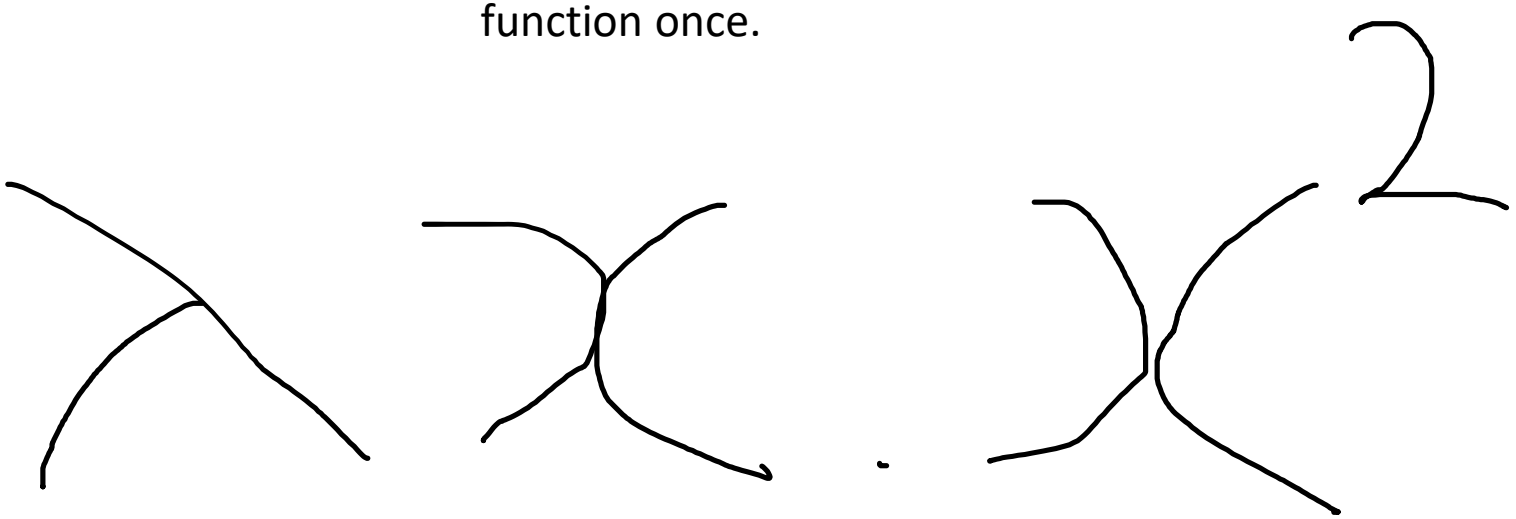
For example, you might think that the notation x^2 is a perfectly good way of representing a function that squares a number, but in fact, it's an expression that represents the result of squaring x, which leaves open the question of what x represents.

A METHOD OR FUNCTION IS JUST binary numbers (representing instructions) stored somewhere in the computer's memory. Considered as a long string of zeros and ones, a subroutine doesn't seem all that different from a data value such as, for example, as an integer, a string, or an array, which is also represented as a string of zeros and ones in memory. We are used to thinking of subroutines and data as very different things, but inside the computer, a subroutine is just another kind of data. Some programming languages make it possible to work with a subroutine as a kind of data value. In Java 8, that ability was added to Java in the form of something called lambda expressions.

We can define a function with *x* as a dummy parameter:

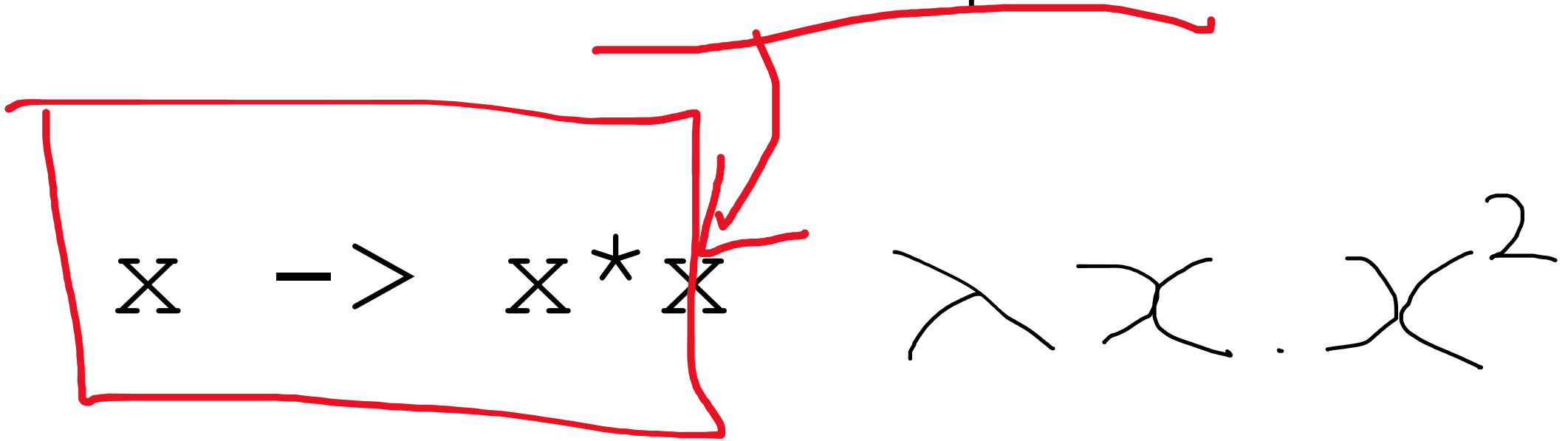
```
static double square( double x ) {
        return x*x;
}
```

but to do that, we had to name the function *square*, and that function becomes a permanent part of the program—which is overkill if we just want to use the function once.



Alonzo Church introduced the notation lambda(x).x^2 to represent "the function of x that is given by x^2"

Java lambda expressions



Means: "the function of x that is given by x^2"

Java lambda expressions

```
x  ->  x*x
```

The operator -> is what makes this a lambda expression

Java lambda expressions

```
sqrt = x -> x*x
```

```
x -> x*x
```

```
sqrt(42)
```

The operator -> is what makes this a lambda expression

```
// to sort a list of apples in inventory based on their weight

// Before:
Collections.sort( inventory, new Comparator<Apple>() {
    public int compare (Apple a1, Apple a2) {
        return a1.getWeight().compareTo( a2.getWeight() );
    }
});

// it is equivalent to
Comparator<Apple> byWeight = new Comparator<Apple>() {
    public int compare (Apple a1, Apple a2) {
        return a1.getWeight().compareTo( a2.getWeight() );
    }
};
Collections.sort( inventory, byWeight );

// After (with lambda expressions):
Comparator<Apple> byWeight =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo( a2.getWeight() );
Collections.sort( inventory, byWeight );

// or
Collections.sort( inventory,
    (Apple a1, Apple a2) -> a1.getWeight().compareTo( a2.getWeight() )
);
```

Functional Interfaces

To know how a subroutine can be legally used, you need to know its name, how many parameters it requires, their types, and the return type of the subroutine.

A functional interface specifies this information about one subroutine. A functional interface is similar to a class, and it can be defined in a .java file, just like a class. However, its content is just a specification for a single subroutine.

A functional interface is an interface that contains only one abstract method.

They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.

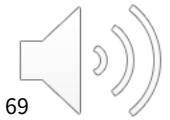
Java comes with a number of built in functional interfaces that are suitable for many situations…

```
public interface FunctionR2R {
    double valueAt( double x );
}
```

This code would be in a file named *FunctionR2R.java*. It specifies a function named *valueAt* with one parameter of type double and a return type of double.

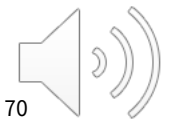
```
public interface ArrayProcessor {
    void process( String[] array, int count );
}
```

This one is called ArrayProcessor and specifies a method process that takes a String[] array and an int



Some different ways of writing lambda expressions

```
Runnable noArguments = () -> System.out.println("Hello World");

ActionListener oneArgument = event -> System.out.println("button clicked");

Runnable multiStatement = () -> {
   System.out.print("Hello");
   System.out.println(" World");
};

BinaryOperator<Long> add = (x, y) -> x + y;

BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;
```

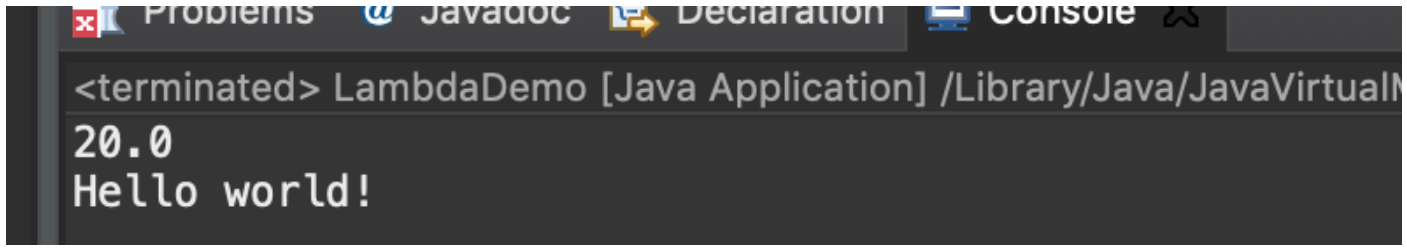


Built in Functional Interfaces

The name of a functional interface is a **type**, just as *String* and double are types.

Here are are some examples

```
// interface binary operator has two arguments and
// requires to implement a method .apply()
BinaryOperator<Double> add = (x,y) -> {return x+y;};
System.out.println(add.apply(10.0, 10.0));

// another useful interface is the runnable interface which
// takes no arguments but runs the statements provided
// it has the method to implement .run()
Runnable noArgs = () -> System.out.println("Hello world!");
noArgs.run()
```



`java.util.function.Function`

Interface Function<T,R>

```
// the interface
public interface Function<T,R> {
        public <R> apply(T parameter);
}

// We can implement with a class
public class AddOne implements Function<Long, Long> {
@Override // more on annotations later
public Long apply(Long aLong) {
        return aLong + 1;
        }
}

// Or with a lambda expression
Function<Long, Long> adder = (value) -> value + 1; Long
resultLambda = adder.apply((long) 8);
System.out.println("resultLambda = " + resultLambda);
```

The Function interface represents a function (method) that takes a single parameter of type T and returns a single value of type R.

Type Parameters:
T - the type of the input to the function
R - the type of the result of the function

```
// in which case to use instantiate the
// interface (and return 2):
LambdaDemo x = new LambdaDemo();
Function<Long, Long> adder = x.new AddOne();
Long result = adder.apply((long) 1);
System.out.println("result = " + result);
```

Table 2-1. Important functional interfaces in Java

Interface name	Arguments	Returns	Example
`Predicate<T>`	`T`	`boolean`	Has this album been released yet?
`Consumer<T>`	`T`	`void`	Printing out a value
`Function<T,R>`	`T`	`R`	Get the name from an `Artist` object
`Supplier<T>`	None	`T`	A factory method
`UnaryOperator<T>`	`T`	`T`	Logical not (`!`)
`BinaryOperator<T>`	`(T, T)`	`T`	Multiplying two numbers (`*`)

Type inference

```
Predicate<Integer> atLeast5 = x -> x > 5;
```

The predicate interface in code, generating a boolean from an Object

```
public interface Predicate<T> {
    boolean test(T t);
}
```

A more complex type inference example

```
BinaryOperator<Long> addLongs = (x, y) -> x + y;
```

```
BinaryOperator<Double> add = (x,y) -> x+y;
```

Code doesn't compile due to missing generics

```
BinaryOperator add = (x, y) -> x + y;
```

This code results in the following error message:

```
Operator '&#x002B;' cannot be applied to java.lang.Object, java.lang.Object.
```

Why use lambda expressions

- Mainly in cases where we are interested in using the behaviors (i.e. methods) of an interface rather than the rest of a class
- For example we have seen the comparator object in the last class
- With a "Comparator" object we usually just want the compareTo() method…

Making Static Methods into Lambdas

Lambdas Made from Static Methods

```
IntFunction<String> intToString = Integer::toString;
ToIntFunction<String> parseInt = Integer::valueOf;
```

Making Constructors into Lambdas

Lambdas Made from a Constructor

```
Function<String,BigInteger> newBigInt = BigInteger::new;
```

Method Reference Operator ::

Making Static Methods into Lambdas

Lambdas Made from Static Methods

```
IntFunction<String> intToString = Integer::toString;
ToIntFunction<String> parseInt = Integer::valueOf;
```

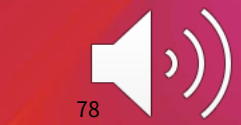
Making Constructors into Lambdas

Lambdas Made from a Constructor

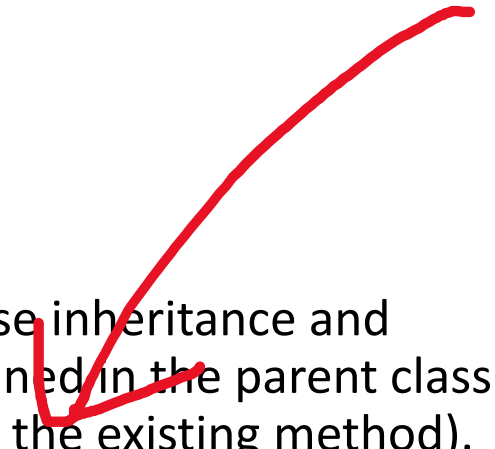
```
Function<String,BigInteger> newBigInt = BigInteger::new;
```

Java Code Annotations

Week 4 Presentation 2

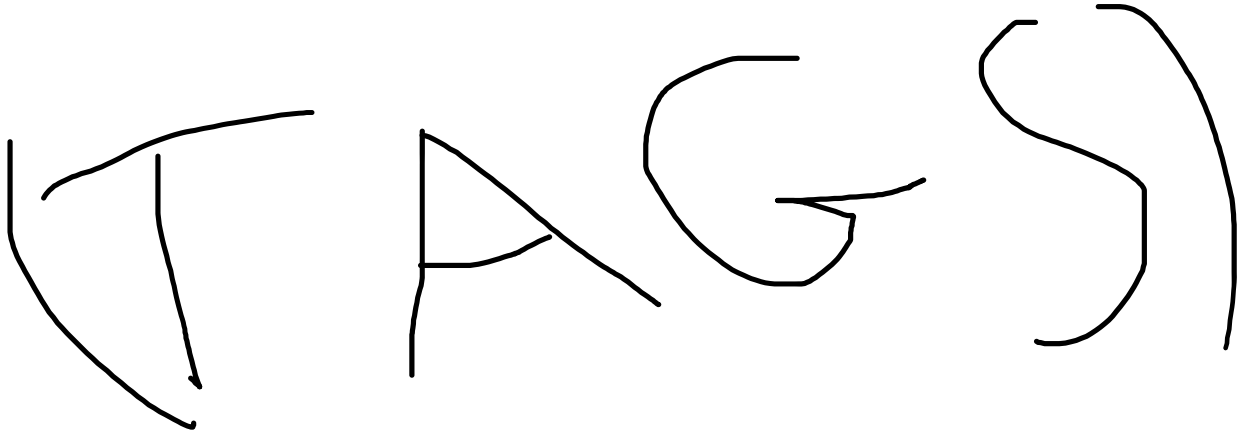


Annotations



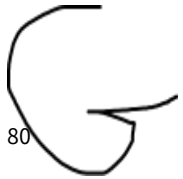
- You may have noticed some annotations already; it's common when you use inheritance and extend an abstract class. Here you redefine in the child class a method defined in the parent class to precede the child class definition with @Override (which means *replace* the existing method).
- This is an annotation, which is completely optional but warns javac of your intent. If you mistype the name, it will enable javac to detect an error if there is no such method in the parent class.

Annotations



- Completely optional
  Change nothing to what the program does
- Help Javac – or the program
- Much used by code-generating tools

As annotations can be accessed by programs, many tools that generate code – for tests, for instance – use annotations to collect information they cannot get otherwise.



Annotations

Annotations can take different forms, from the simple marker to some kinds of function calls that are outside the program itself.

- Marker e.g. @Override
- Single parameter
- Multiple parameters

@Select (sql ="SELECT userId, firstName,…

Metadata

= DATA about the CODE

Metadata is a big concern in real life. Companies consider programs as assets, on which several generations of developers can work, which must be written in an easy-to-comprehend, standard way, and well documented. Metadata allows, among many other things, to industrialize code production and to standardize everything.

Standard Annotations

- @Override
- @Deprecated
- @SuppressWarnings

Java provides three standard annotations, which are all a way to give hints to javac.

= old / obsolete

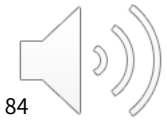
e.g.
@SuppressWarnings({"deprecation", "unchecked"})

2 Annotations were added in Java 7

- @SafeVarargs
- @FunctionalInterface

"Varargs" stands for "Variable [number of] Arguments"

FunctionalInterface marks a functional interface as we saw can be assigned to anonymous functions

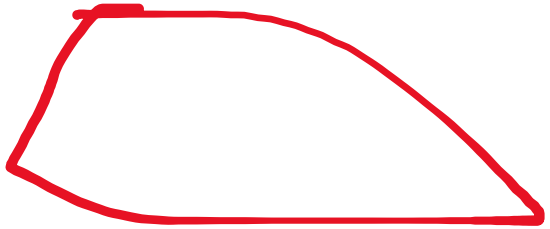


Making Annotations

- You can create your own annotations!

Declared as interfaces

```
import java.lang.annotation.*;

public @interface MyAnnotation {
}
```



Annotation-based tools use their own set of annotations, which you just need to import before using.

Making Annotations

- As annotations are a bit special (it's a kind of program in the program) they are constrained by a number of rules.

- Methods should not have any parameters.
- Methods declarations should not have any throws clauses.
- *Return type must be one of:*
  - primitive type, String, enum, Class or an array of one of these

https://docs.oracle.com/javase/tutorial/java/annotations/

Custom Annotations Example: Structured Documentation

```
class SomeClass {
// Created by S Faroult // Creation date: 21/03/2017
// Revision history:
//
// 24/05/2017–Constructor with String parameter
// 26/02/2018 – toString() rewritten
```

What can you use annotations for in practice? Any Software Development Manager dreams of seeing comments like this. But every developer will not write them, and those who do may use a different format.

Custom Annotations Example: Structured Documentation

```
import java.lang.annotation.*;

public @interface ClassDoc {
        String author();
        String created();
        String[] revisions();
}
```

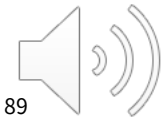
ClassDoc.java

Annotations could help turning readable but unparsable comments into usable data…

Custom Annotations Example: Structured Documentation

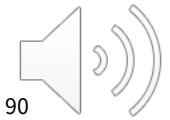
```
@ClassDoc (          author="S Faroult",
                     created="21/03/2017",
                     revisions={"24/05/2017 – Constructor
            with String parameter",
                     "26/02/2018 – toString() rewritten"})
class SomeClass {
```

As the annotation is defined and checked by javac it becomes possible to ensure a standard way for documenting the code. This information can then be retrieved to document programs (such as with Javadoc – will cover later).



Meta Annotations (Annotations about Annotations)

- **@Retentions:** says whether the annotation is available to javac or available at runtime.
- **@Documented**: Make it appear in the docs generated by Javadoc tool.
- **@Target**: What it applies to: Constructor, Method, Parameter…
- **@Inherited**: Passed to child classes (false by default).
- **@Repeatable**: Can be applied multiple times.



Testing

- JUNIT generates tests for checking your programs. Frameworks are software tools that try to generate automatically the boring bits of a program (which are often a lot of copy-and-paste).

https://junit.org/junit5/

@Test

@BeforeClass – Run once before any of the test methods in the class, public static void
@AfterClass – Run once after all the tests in the class have been run, public static void
@Before – Run before @Test, public void
@After – Run after @Test, public void
@Test – This is the test method to run, public void

Reflection

Week 4 Presentation 3

Reflection

- Generally speaking, "reflection" is your program asking the JVM what it knows about it – and the JVM knows a lot of things.
- As all this happens of course while the program is running, it allows for a lot of on-the-fly operations that would be impossible with a compiled program written in C, for instance.
- Reflection is considered rather advanced programming, but some of its features are frequently used, for instance with JDBC which is the standard Java way to access a database.

Reflection

- Works because of the JVM. Once again, it only works because of the JVM
- The JVM stores the description of the classes when it loads them
- The loading subsystem needs to read a lot of information to make the program runnable, and this information is stored and made available when the program runs

Reflection

- The JVM stores objects, of class **Class**, that describe every class used in the application

- Class called **Class**
  - *Metadata*
- The objects represent classes in the running application

- Class objects have no constructor – they are built by the JVM

Class objects

- There are two ways two retrieve class information from the JVM.

ClassName obj = new ClassName();

Method inherited from object

1. Method inherited from the object:

   The get class method of the object: obj.getClass()

2. The .class attribute when there is no object: ClassName.class

Static version

no constructor – built by the JVM

Uses of Reflection #1: Getting Class Names

For instance, you can retrieve class names.

```
class OuterClass {
        private int dummy;
        OuterClass(){}
}
public class MyClass {

        class InnerClass {
                private int dummy;
                InnerClass(){}
        }
public static void main(String[] args) {
        OuterClass obj = new OuterClass();
        System.out.println(obj.getClass().getName());
        System.out.println(InnerClass.class.getName());
}
```

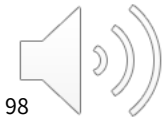
```
$
$ java MyClass
OuterClass
MyClass$InnerClass
$
```

Uses of Reflection #2: Locating Files used by Your Program

- One common problem is locating files used by your program – the properties file to start with if there is one.
- Location of files read by your program
  - parameter file
  - data file multimedia, etc

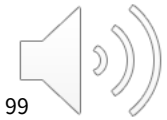
When people click on an icon to launch your program, the idea of "current directory" becomes extremely hazy. If you want to start by reading a properties file, of if you want to display the logo of your company (an image) while initialization is going on, where should you look?

The default directory for installing programs varies from system to system (and don't forget that a Java application can run on Windows as well as on Linux or Mac OSX), and additionally users often have the option of installing software elsewhere than the default location. Your only hope to find out is to get it when the program runs.



Uses of Reflection #2: Locating Files used by Your Program

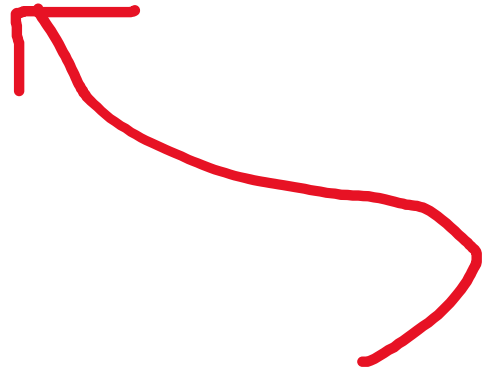
- As the loader knows where it got the .class from, you can just ask the JVM.
- **Solution** get location at runtime



Uses of Reflection #2: Locating Files used by Your Program

- As the loader knows where it got the .class from, you can just ask the JVM.
- **Solution** get location at runtime

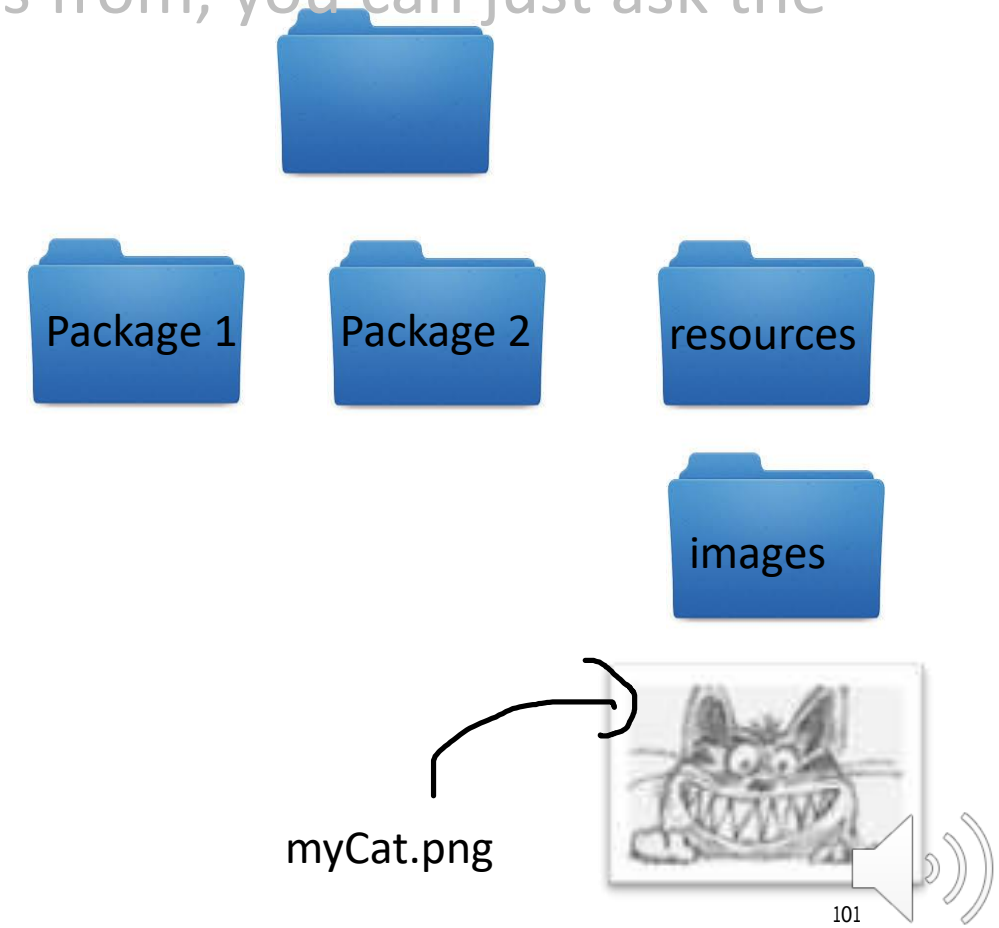
file:/Users/... ... .../Reflection.class



```
public class Reflection {

	public static void main(String[] args) {
		System.out.println(Reflection
				.class
				.getClassLoader() .getResource("Reflection.class") .toSt

	}
}
```

Uses of Reflection #2: Locating Files used by Your Program

- As the loader knows where it got the .class from, you can just ask the JVM.
- **Solution** get location at runtime




```
URL url = this
            .getClass()
             .getClassLoader()
             .getResource("resources/images/myCat.png");
```

Uses of Reflection #3: Reading Annotations

- We saw in the previous presentation that annotations could be read by a program, it's through reflection.
- Done by many tools such as JUNIT and more we will see later.

Uses of Reflection #3: Reading Annotations

- We saw in the previous presentation that annotations could be read by a program, it's through reflection.
- Done by many tools such as JUNIT and more we will see later.

- By default annotations are **NOT** visible at runtime so we make them so as follows

@Retention(RetentionPolicy.RUNTIME) `

Uses of Reflection #3: Reading Annotations

- By default annotations are **NOT** visible at runtime so we make them so as follows

@Retention(RetentionPolicy.RUNTIME) `

Remember that @Retention() is a meta-annotation, an annotation that applies to annotations.

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
public @interface ClassDoc {
        String author();
        String created();
        String[] revisions();
}
```

Uses of Reflection #3: Reading Annotations

- If SomeClass is annotated with an annotation available at runtime ...

```
@ClassDoc(
        author="S Faroult",
        created="21/03/2017",
        revisions={"24/05/2017 – Constructor with String parameter",
                "26/02/2018 – toString() rewritten"})
class SomeClass {
}
```

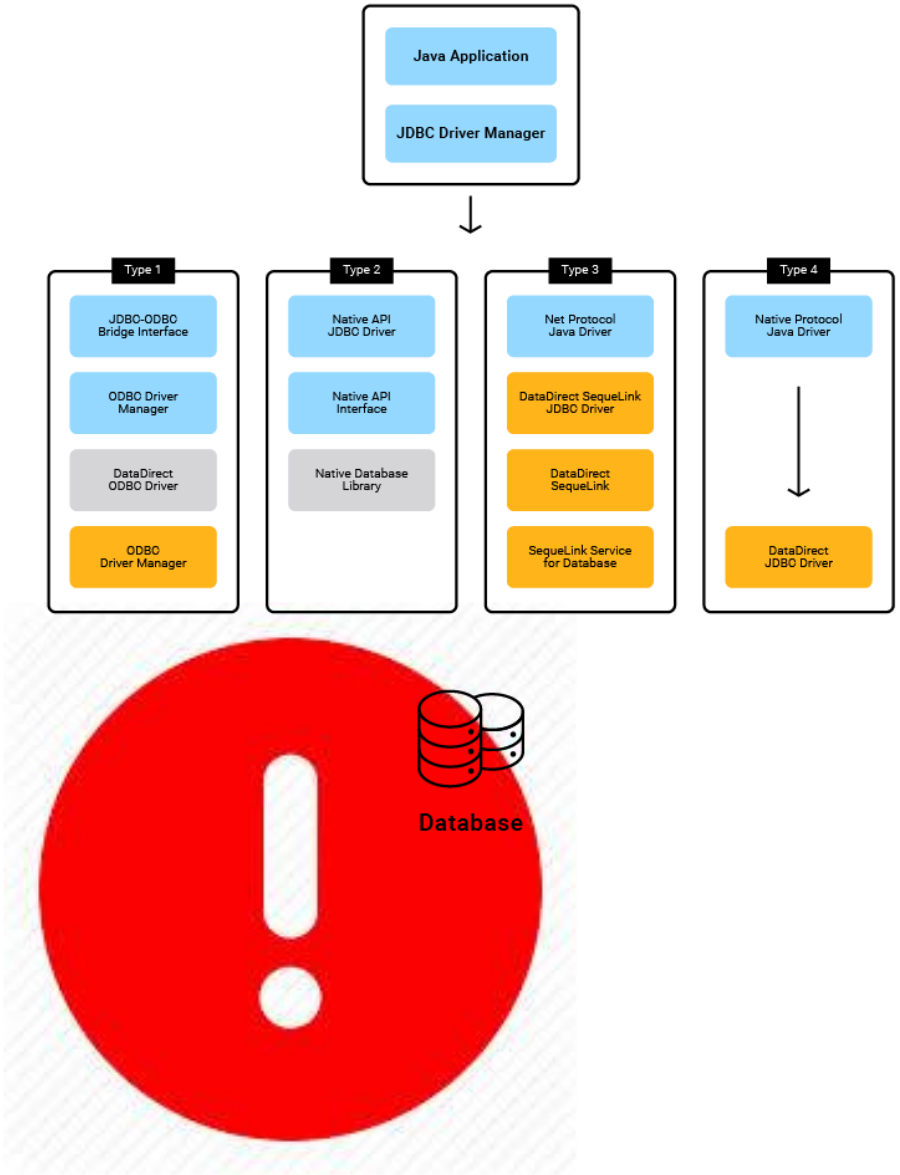
Note: it must be recompiled if ClassDoc is changed for getAnnotations to see it…

Uses of Reflection #3: Reading Annotations

- Then annotations gets it…

```
import java.lang.annotation.Annotation;
public class ReadingAnnotations {

        public static void main(String[] args) {
                Annotation[] annotations = SomeClass .class
                                                .getAnnotations();
                for (Annotation annot: annotations) {
                        System.out.println(annot.toString());
                }
        }
}
```

```
$ java ReadingAnnotations
@ClassDoc(author=S Faroult, created=21/03/2017
revisions=[24/05/2018 - Constructor with Strin
parameter, 26/02/2018 - toString() rewritten])
$
```

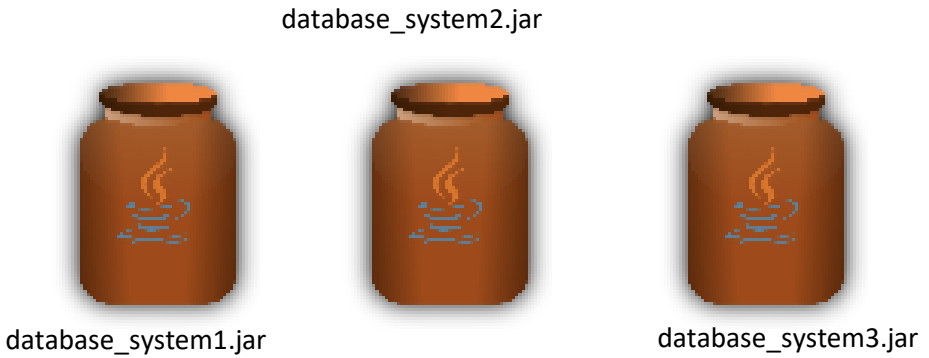


Uses of Reflection #4: Dynamically loading a class

- This is another very useful application of reflection
- Much used for "drivers" of hardware
- Because of the many different standards (National, International, Proprietary, etc) identical functionality is often achieved by different classes, that work with one special piece of hardware or software.

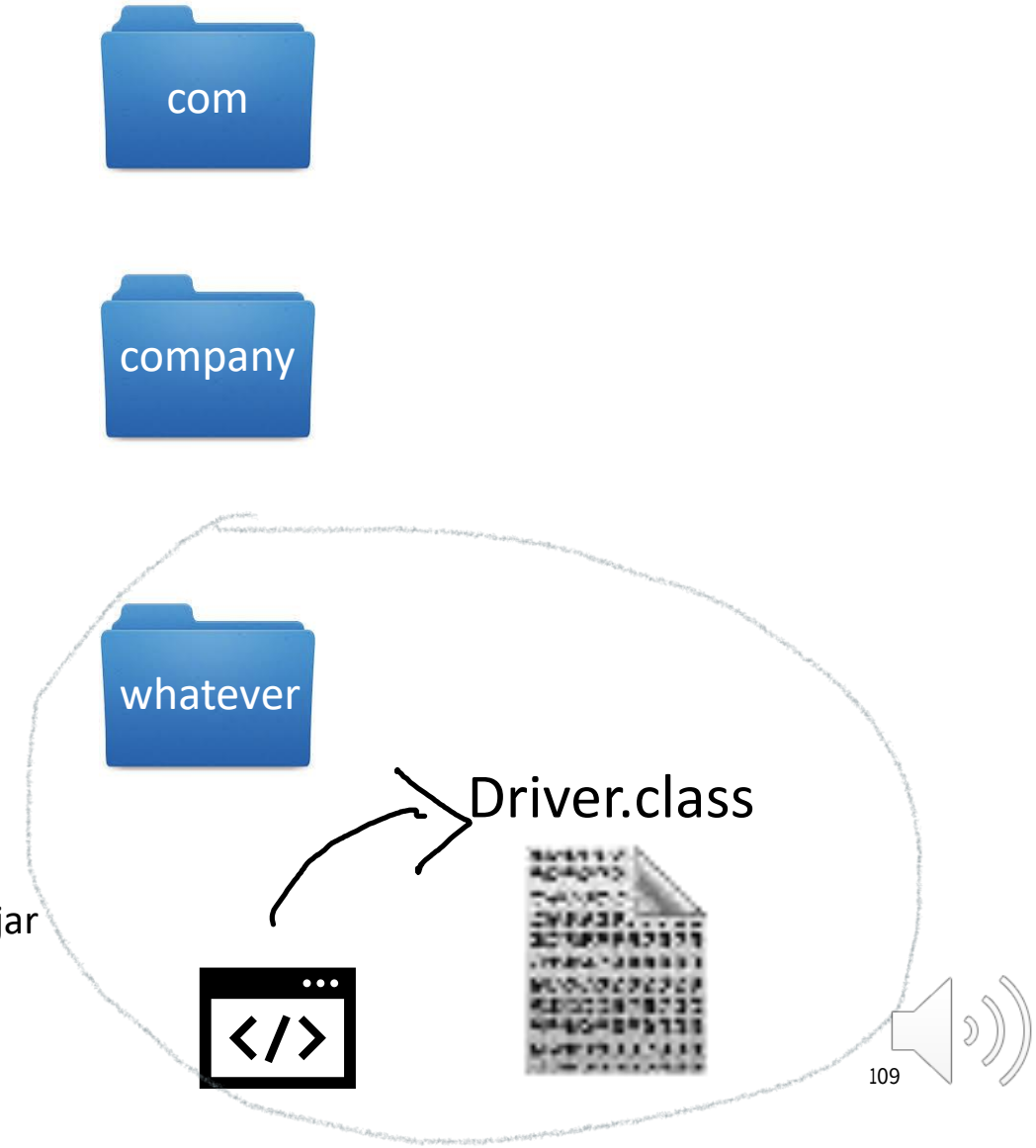
Uses of Reflection #4: Dynamically loading a class

- This is particularly useful with database access. Although there is a common language for accessing databases, database providers supply (as java archives) classes that implement the required methods to talk to THEIR system.
- More later when we discuss JDBC.



Uses of Reflection #4: Dynamically loading a class

Usually the driver has a long complicated name to ensure that there is no conflict (two different drivers cannot have the same name).

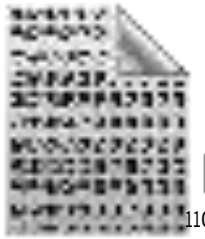


Uses of Reflection #4: Dynamically loading a class

On the CLASSPATH

If the name of the .jar file is included in the CLASSPATH (where the loader looks for .class files), then the program can load the driver of its choice.


```
Class c = Class.forName("com.company.whatever.Driver");
Driver d = (Driver) c.newInstance();
```



Exercise: Download and run Squirrel SQL

There is a Java graphical tool called Squirrel SQL that uses this to let you query almost any database system, as long as you have the suitable .jar file added to your CLASSPATH. You can switch between very different systems.

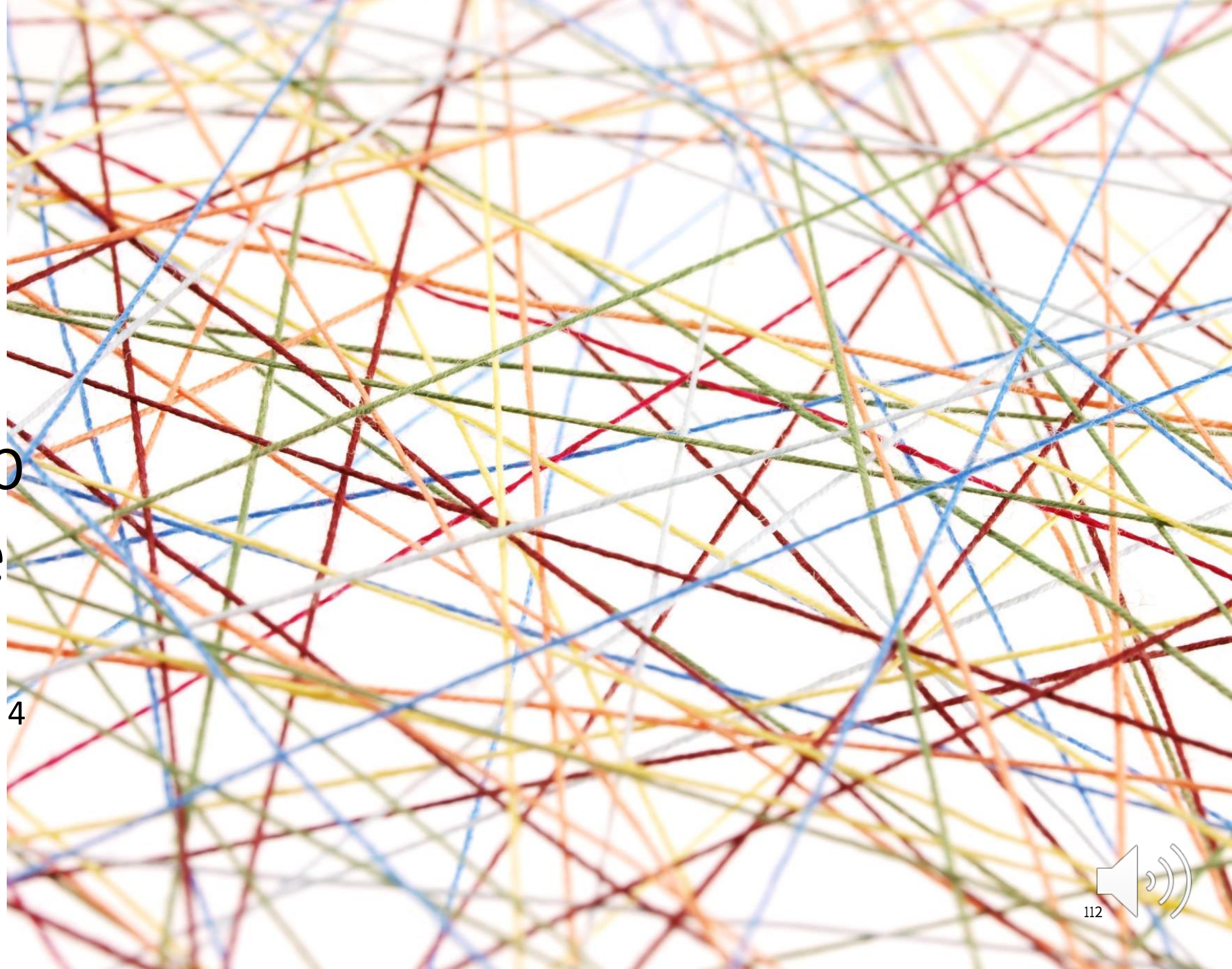
Download link here: http://squirrel-sql.sourceforge.net

Follow to the tutorial here to view some databases:

http://squirrel-sql.sourceforge.net/kulvir/tutorial.html

HashMap Example

Week 4 Presentation 4



Maps

Table 5 Working with Maps

`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<>();`	Use a `HashMap` if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.
`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

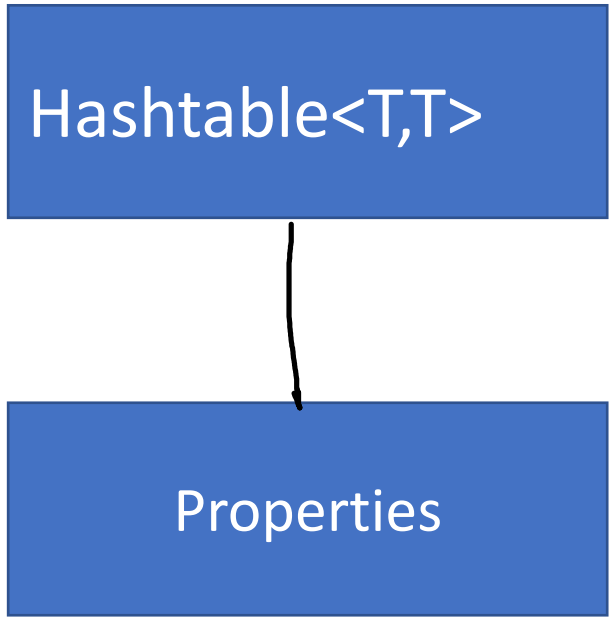
`for (String key : scores.keySet()) {`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

Properties

- One interesting (and very useful) example of hash tables is the Properties class, which associates two strings.
- java.util.Properties

- java.lang.Object
  - java.util.Dictionary<K,V>
    - java.util.Hashtable<Object,Object>
      - java.util.Properties

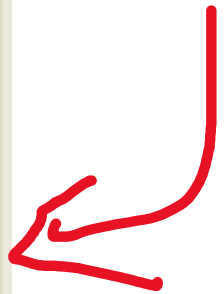


Key: String

Value: String

```
#Location of data files
data_dir = C:\Users\Public\Data
# Remote server
server = 192.168.1.214
# Theme name
theme = Funky
```

preferences.cnf



Example 1

Properties

When you install a piece of software on your computer, you are usually asked a lot of questions, such as where you want to install the program, the location of other resources, possibly a theme. All this information is stored in a .ini, .conf or whatever file. Each parameter is a name associated with a value. Properties objects deal with these files, can read them, write them, and ignore for instance lines starting with #.

You can define default values. Calling prop.get() would return null for a value not read from the file, prop.getProperty() returns the default if there is one.

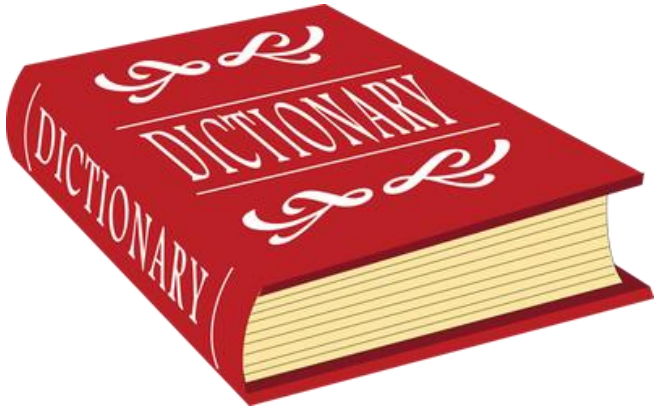
```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;

public class PropertiesEx {

    public static void main(String args[]) {

        Properties defprop = new Properties();
        defprop.put("data_dir", ".");
        defprop.put("theme", "classic");
        Properties prop = new Properties(defprop);

        try (BufferedReader conf = new BufferedReader(new FileReader("preferences.cnf"))) {

            prop.load(conf);

        }catch (IOException e) { // Ignore
            System.err.println("Warning: using default preferences");
        }

        // Display the preferences
        System.out.println(prop.getProperty("data_dir"));
        System.out.println(prop.getProperty("theme"));
    }

}
```

Try with resources. Remember?

Example 2

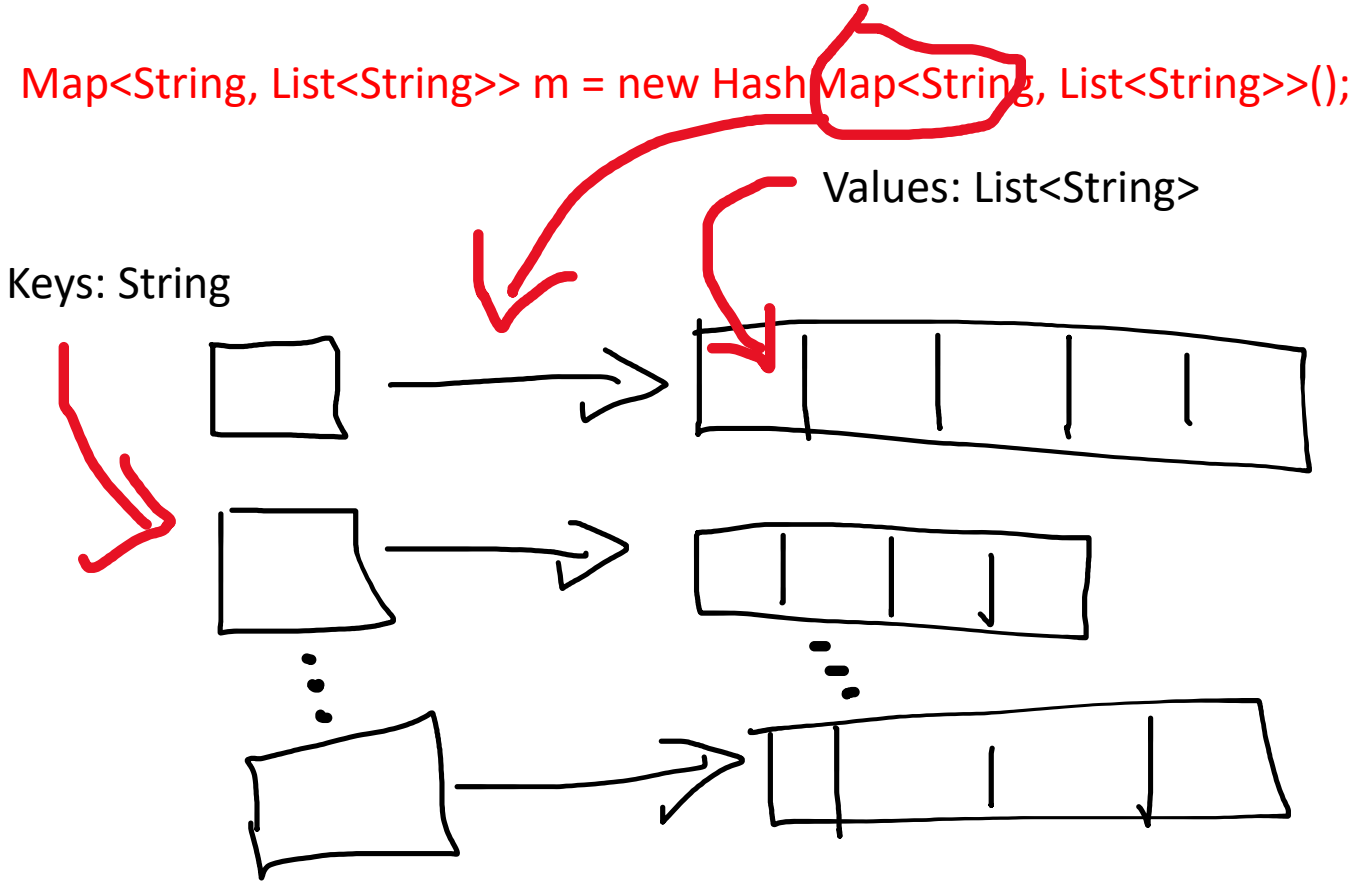
A *multimap* is like a Map but it can map each key to multiple values.

Problem: how can we find all the anagrams of every word in a dictionary?



Anagrams are words that are created by rearranging the letters in a word – for example “late” is an anagram of “tale”.

A *multimap* is like a Map but it can map each key to multiple values.



```java
import java.util.*;

public class Anagrams {
    public static void main(String[] args) {
        int minGroupSize = Integer.parseInt(args[1]);

        // Read words from file and put into a simulated multimap
        Map<String, List<String>> m = new HashMap<String, List<String>>();

        try {
            Scanner s = new Scanner(new File(args[0]));
            while (s.hasNext()) {
                String word = s.next();
                String alpha = alphabetize(word);
                List<String> l = m.get(alpha);
                if (l == null)
                    m.put(alpha, l=new ArrayList<String>());
                l.add(word);
            }
        } catch (IOException e) {
            System.err.println(e);
            System.exit(1);
        }

        // Print all permutation groups above size threshold
        for (List<String> l : m.values())
            if (l.size() >= minGroupSize)
                System.out.println(l.size() + ": " + l);
    }

    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

There is a standard trick for finding anagram groups: For each word in the dictionary, alphabetize the letters in the word (that is, reorder the word's letters into alphabetical order) and put an entry into a multimap, mapping the alphabetized word to the original word.

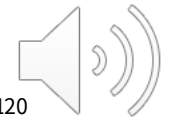
For example, the word *bad* causes an entry mapping *abd* into *bad* to be put into the multimap.

A moment's reflection will show that all the words to which any given key maps form an anagram group. It's a simple matter to iterate over the keys in the multimap, printing out each anagram group that meets the size constraint.

```
> java Anagrams "../dictionary.txt"
5
```

5: [dates, sated, stade, stead, tsade]
6: [dearths, hardest, hardset, hatreds, threads, trashed]
6: [dater, derat, rated, tared, trade, tread]
5: [entasis, nasties, sestina, tansies, tisanes]
5: [actors, castor, costar, scrota, tarocs]
5: [delist, idlest, listed, silted, tildes]
5: [intreat, iterant, nattier, nitrate, tertian]
9: [anestri, antsier, nastier, ratines, retains, retinas, retsina, stainer, stearin]

See: https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html



Example 3

Another example uses a Tokenizer class that counts words from a text file one by one, ignoring space and punctuation.

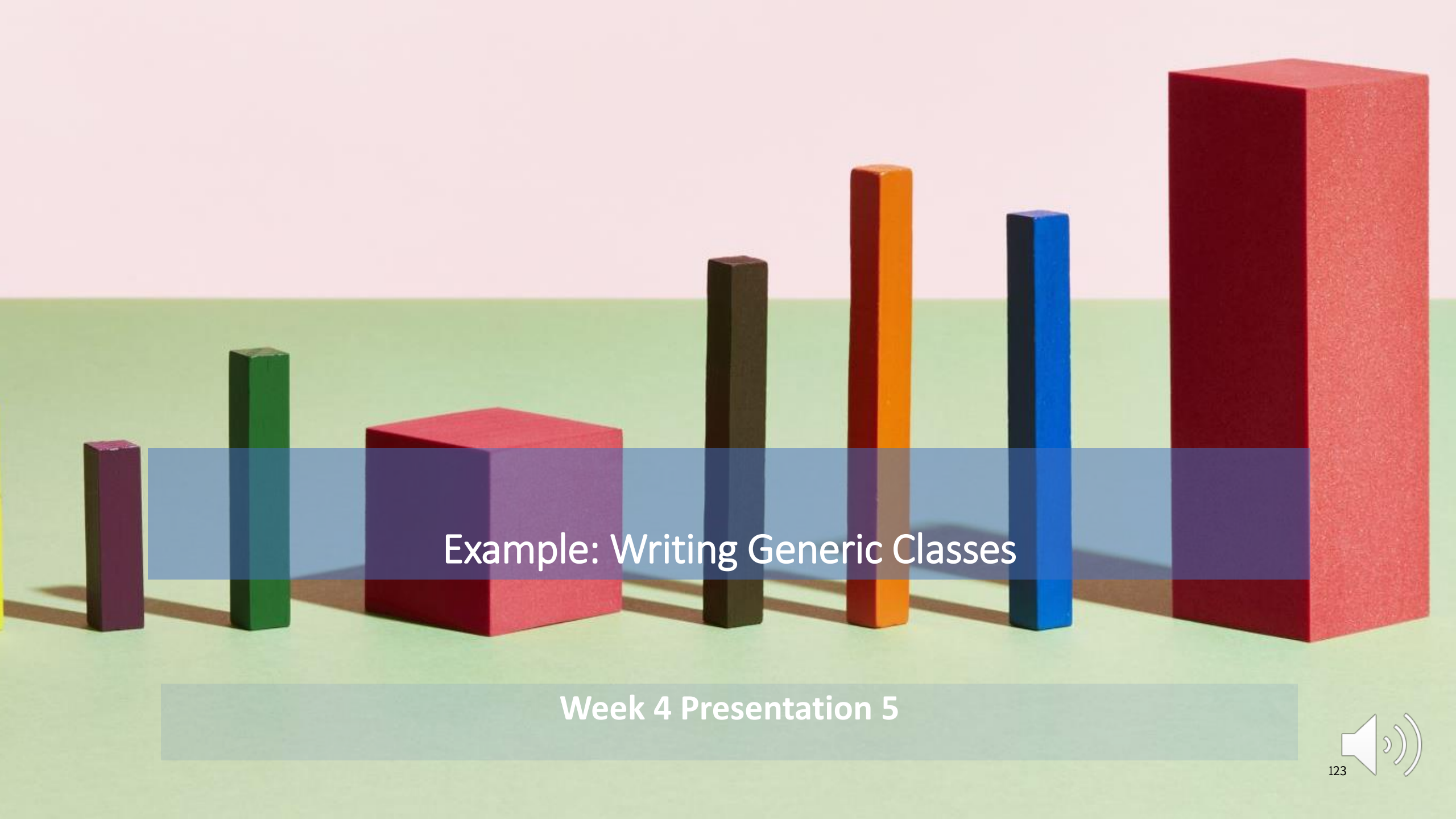
Goal: finding the 10 most used words in a speech.

We need to associate with each word a counter (so, we need a Map<String,Integer>). If we don't know the word, we store it with "1". Otherwise, we retrieve the counter, increase its value by one, and store it back.

When we have counted words, we need to find the 10 most used – we can use a map (associating a number of occurrences to a list of words, as there may be ties) but we also need some ordering. We need to go through our hash map and store its objects in, for instance, a TreeMap<Integer,TreeSet<String>>. Then we can iterate on the tree map and retrieve the most common words.

The result is usually very disappointing, because in an English speech the most common words are likely to be "the", "is", "a", and so forth. Those words are not very significant words and are usually called "stop words" (search engines on the Internet ignore them).
What we need to do is have a list of stop words, read it into an easily searchable structure such as a tree, and start counting words only when we cannot find them in this list of not important words. It gives a completely different vision of a speech.

Exercise: implement this algorithm.



Example: Writing Generic Classes

Week 4 Presentation 5

A Queue as a Linked List

To ensure that the only operations that are performed on the list are the queue operations **enqueue**, **dequeue**, and **isEmpty**, we can create a new class that contains the linked list as a private instance variable.

```
class QueueOfStrings {
   private LinkedList<String> items = new LinkedList<>();
   public void enqueue(String item) {
      items.addLast(item);
   }
   public String dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
}
```

```
class Queue<T> {
   private LinkedList<T> items = new LinkedList<>();
   public void enqueue(T item) {
      items.addLast(item);
   }
   public T dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
}
```

if we want queues of *Integers* or *Doubles* or *Colors* or any other type, we can write a **generic** *Queue* class that can be used to define queues of any type of object.

Generic Class: A Queue as a LinkedList

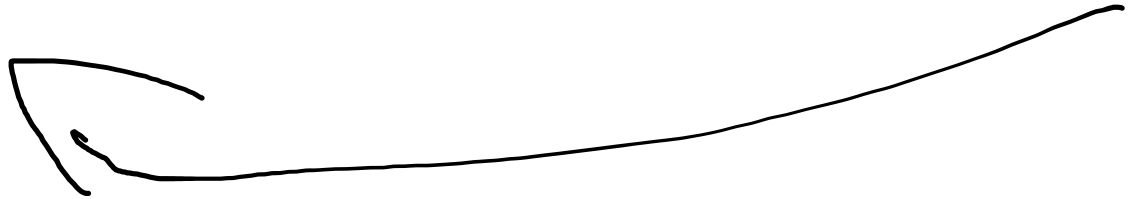
Given this class definition, we can use parameterized types such as *Queue<String>* and *Queue<Integer>* and *Queue<Color>*.

```
class Pair<T,S> {
   public T first;
   public S second;
   public Pair (T a, S b) {  // Constructor.
      first = a;
      second = b;
   }
}
```

can be used to declare variables and create objects such as:

```
Pair<String,Color> colorName = new Pair<>("Red", Color.RED);
Pair<Double,Double> coordinates = new Pair<>(17.3,42.8);
```

It's also easy to define generic classes and interfaces that have two or more type parameters, as is done with the standard interface *Map<K,V>*. A typical example is the definition of a "Pair" that contains two objects, possibly of different types. A simple version of such a class can be defined as the class Pair here



```
/**
 * Returns the number of times that itemToCount occurs in list. Items
 * in the list are tested for equality using itemToCount.equals(),
 * except in the special case where itemToCount is null.
 */
public static int countOccurrences(String[] list, String itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for (String listItem : list)
         if (listItem == null)
            count++;
   }
   else {
      for (String listItem : list)
         if (itemToCount.equals(listItem))
            count++;
   }
   return count;
}
```

For a generic method, the "<T>" goes just before the name of the return type of the method:

```
public static <T> int countOccurrences(T[] list, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for (T listItem : list)
         if (listItem == null)
            count++;
   }
   else {
      for (T listItem : list)
         if (itemToCount.equals(listItem))
            count++;
   }
   return count;
}
```

For a generic method, the "<T>" goes just before the name of the return type of the method:

```
public static <T> int countOccurrences(T[] list, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for (T listItem : list)
         if (listItem == null)
            count++;
   }
   else {
      for (T listItem : list)
         if (itemToCount.equals(list
            count++;
   }
   return count;
}
```

If `wordList` is of type `String[]` and `word` is of type *String*, then

```
int ct = countOccurrences( wordList, word );
```

will count the number of times that `word` occurs in `wordList`.

If `palette` is of type `Color[]` and `color` is of type *Color*, then

```
int ct = countOccurrences( palette, color );
```

will count the number of times that `color` occurs in `palette`.

If `numbers` is a variable of type `Integer[]`, then

```
int ct = countOccurrences( numbers, 17 );
```

will count the number of times that `17` occurs in `numbers`.

```
public static <T> int countOccurrences(Collection<T> collection, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( T item : collection )
         if (item == null)
            count++;
   }
   else {
      for ( T item : collection )
         if (itemToCount.equals(item))
            count++;
   }
   return count;
}
```

The countOccurrences method operates on an array. We could also write a similar method to count occurrences of an object in any