

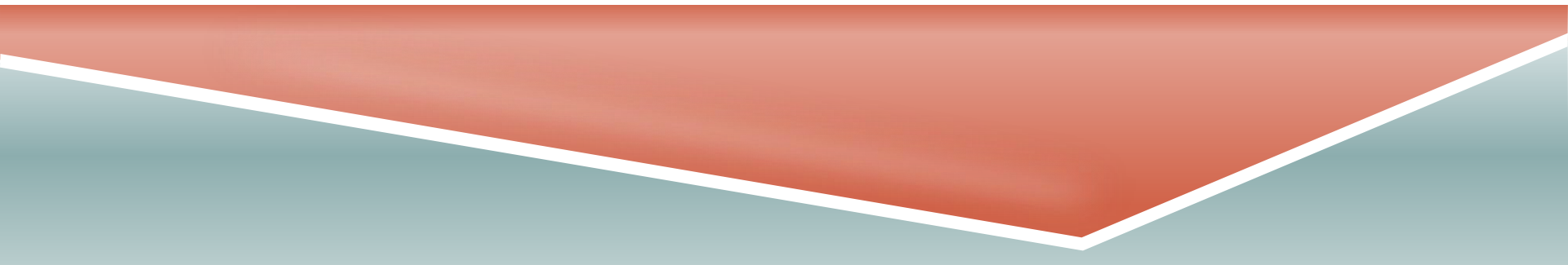
# ENTERPRISE JAVA PROGRAMMING / PROGRAMMING II

MODULES CT545 / CT875, SEMESTER 2, ACADEMIC YEAR 2017-2018

NATIONAL UNIVERSITY OF IRELAND, GALWAY

## *Lecture 2b*

Lecturer: Dr Matthias Nickles



# TOPICS

- ▶ Functors (+ a bit more generic programming)
- ▶ Java Collections Framework

# FUNCTORS

- ▶ We now look at yet another use case for generics...
- ▶ Suppose we want to use `findMax` again (returns the "maximum" element of an array of objects)...
- ▶ If we use the `Comparable` interface here (→ previous lecture), a challenge arises if objects of *one* class have *multiple* different ways of comparing them, and not just a single “natural order”...
- ▶ E.g., two rectangles could be compared in terms of their areas, but also in terms of their perimeters.

# FUNCTORS

```
1 // A simple rectangle class.
2 public class SimpleRectangle
3 {
4     public SimpleRectangle( int l, int w )
5         { length = l; width = w; }
6
7     public int getLength( )
8         { return length; }
9
10    public int getWidth( )
11        { return width; }
12
13    public String toString( )
14        { return "Rectangle " + getLength( ) + " by "
15            + getWidth( ); }
16
17    private int length;
18    private int width;
19 }
```

*Example code in this lecture from Mark A. Weiss: Data Structures and Algorithm Analysis in Java*

```

1 class FindMaxDemo
2 {
3     /**
4     * Return max item in a.
5     * Precondition: a.length > 0
6     */
7     public static Comparable findMax( Comparable [ ] a )
8     {
9         int maxIndex = 0;
10
11         for( int i = 1; i < a.length; i++ )
12             if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
13                 maxIndex = i;
14
15         return a[ maxIndex ];
16     }
17
18     /**
19     * Test findMax on Shape and String objects.
20     */
21     public static void main( String [ ] args )
22     {
23         Shape [ ] sh1 = { new Circle( 2.0 ),
24                           new Square( 3.0 ),
25                           new Rectangle( 3.0, 4.0 ) };
26
27         String [ ] st1 = { "Joe", "Bob", "Bill", "Zeke" };
28
29         System.out.println( findMax( sh1 ) );
30         System.out.println( findMax( st1 ) );
31     }
32 }

```

# FUNCTORS

- ▶ Solution: a generic interface and a so-called *functor*...

# FUNCTORS

- ▶ What is a "functor"...
- ▶ A *functor* (also called "function object") is an object which represents a function.
- ▶ A functor is often used for passing a method *as an argument* to some other method.
- ▶ Typical (and most important) use case for functors: *callbacks*, such as event handlers
- ▶ Also close connection to --> *anonymous classes* (later lectures)
- ▶ Term "functor" isn't standardized, an alternative name for the same concept is "function object"
- ▶ *Remark: in contexts other than Java, the word "functor" might have a different meaning!*

# FUNCTORS

- ▶ Functors are not an entirely new language element, they are just a certain application of plain objects, interfaces and classes.
- ▶ So you just need to apply concepts you know already.
- ▶ You might have seen them informally already in the context of interface Comparable



# FUNCTORS

- ▶ Functors are needed because in Java < 8, functions are not *first-class citizens* - you cannot pass functions (methods) directly as arguments to other functions or store functions directly in variables.
- ▶ Later you will learn about *Lambda Expressions* as a way of solving this problem using some other means. But in the very common Java versions 6 and 7, these don't exist.
- ▶ In Java <=7, we need to "wrap" functions in certain objects in order to pass them around: functors...

# FUNCTORS

- ▶ Functors are typically realized using
  - 1) interfaces with just a single abstract method  $m$ .  
 $m$  is the function represented by the functor.  
and
  - 2) classes which implement this interface.
- ▶ Objects of the classes 2) can then be used as *representatives* of the various incarnations of method  $m$ . They are called “functors”.
- ▶ These objects can then be used, e.g., to pass on these methods to some other method, where the respective variant of  $m$  can then be called.
- ▶ A functor can be realized using, e.g., a so-called *anonymous class* (a class without a name), however, that's just a technicality, not the core idea of functors.

# FUNCTORS

- ▶ An abstract example:
  - ▶ Suppose you create a method `myMethod`. Within the body of this method, you would like to call another method `mx`.
  - ▶ Assume there are multiple options for method `mx`, and you don't want to restrict `myMethod` to only one of these variants. Instead, you want to let the *caller* of `myMethod` decide which variant of `mx` your method should call.
  - ▶ Solution: you introduce an additional parameter of `myMethod` for passing a functor to `myMethod`.
  - ▶ Each time you would call `mx` in `myMethod`, instead you call the method within the functor.

# FUNCTORS

- ▶ How does all this apply to our first example: comparing objects...?
- ▶ We want to use functors to be able to pass on *various* comparison-methods to the `findMax`-method
- ▶ Firstly, we need to specify the signatures of all those "comparison" functor methods. Luckily, the signature is the same for all of them. We do this simply by defining an abstract method in interface `Comparator`:

```
abstract int compare(AnyType lhs, AnyType rhs)
```
- ▶ Implementations of this abstract method return -1, 0 or 1, depending on whether `lhs` is less/equal/larger than `rhs`
- ▶ `AnyType` should be a type parameter (placeholder for the concrete types of the two objects `lhs` and `rhs`), so we need to make the interface generic...

# FUNCTORS

- ▶ Let's look at the `Comparator` interface...
- ▶ Its almost identical with Java's built-in generic interface `Comparable`...

# FUNCTORS

```
1 package weiss.util;
2
3 /**
4  * Comparator function object interface.
5  */
6 public interface Comparator<AnyType>
7 {
8     /**
9      * Return the result of comparing lhs and rhs.
10     * @param lhs first object.
11     * @param rhs second object.
12     * @return < 0 if lhs is less than rhs,
13     *         0 if lhs is equal to rhs,
14     *         > 0 if lhs is greater than rhs.
15     */
16     int compare( AnyType lhs, AnyType rhs );
17 }
```

# FUNCTORS

- ▶ Here is an example class which implements this interface. The objects of this class can be directly used as functors - it's really that simple!
- ▶ The functor's class has the sole purpose of specifying one (among several) particular way to compare objects of type `SimpleRectangle`. It doesn't declare any other methods or fields.

```
1 class OrderRectByWidth implements Comparator<SimpleRectangle>
2 {
3     public int compare( SimpleRectangle r1, SimpleRectangle r2 )
4         { return( r1.getWidth() - r2.getWidth() ); }
5 }
6
```

- ▶ If there are further ways of comparing objects of this type, we would just need to define further classes which also implement `Comparator<SimpleRectangle>` (each with a different implementation of the `compare`-method)

# FUNCTORS

- ▶ Now we can take any of these functors to tell method `findMax` about how concretely it should compare two objects...
- ▶ To this end, we need to pass the respective functor to `findMax` as an argument, so that `findMax` can call it in order to compare objects with each other
- ▶ The functor interface is generic, so when our new version of `findMax` is called, we need an instance (concrete type) for the type parameter/type variable...
- ▶ We can achieve this, e.g., by making the new `findMax` a static generic method. This way, when calling `findMax`, the concrete type of the objects we want to compare is automatically figured out by Java...



# FUNCTORS

- ▶ Each time when `findMax` is called, it calls the specific comparison method in the respective functor.
- ▶ Remark: functors are not required to be based on generics or to use interfaces. Every object whose purpose it is to “encapsulate” a function can be seen as a functor!
- ▶ It is also not required to use a functor in a generic method - that’s just a little trick we use here to discover the type of the objects which want to compare with each other (see next slide for the code)

# FUNCTORS

The new `findMax`, using the generic functor interface `Comparator`:

```
1 public class Utils
2 {
3     // Generic findMax with a function object.
4     // Precondition: a.length > 0.
5     public static <AnyType> AnyType
6     findMax( AnyType [ ] a, Comparator<
7             {
8         int maxIndex = 0;
9
10        for( int i = 1; i < a.length; i++ )
11            if( cmp.compare( a[ i ], a[ maxIndex ] ) > 0 )
12                maxIndex = i;
13
14        return a[ maxIndex ];
15    }
16 }
```

type parameter of static generic  
method findMax

Parameter  
cmp accepts a  
functor

call of the comparison  
method defined  
"inside" the functor

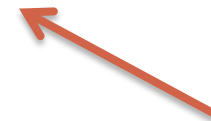
# FUNCTORS

- ▶ Remember from the previous lecture that `AnyType` is the type parameter of this generic static method
- ▶ `AnyType` is a placeholder for a concrete type. E.g., `SimpleRectangle`
- ▶ Remember further that in contrast to using generic classes or generic interface, you do not need to specify this concrete type when you use a *static generic method* like `findMax`. Java figures it out by itself.
- ▶ However, if the concrete type should be, e.g., `Ellipse`, you would need to provide a functor which is an instance of a class which implements `Comparator<Ellipse>` instead of `Comparator<SimpleRectangle>`!

# FUNCTORS

Example for a call of our new method `findMax`:

```
6
7 public class CompareTest
8 {
9     public static void main( String [ ] args )
10    {
11        SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
12        rects[ 0 ] = new SimpleRectangle( 1, 10 );
13        rects[ 1 ] = new SimpleRectangle( 20, 1 );
14        rects[ 2 ] = new SimpleRectangle( 4, 6 );
15        rects[ 3 ] = new SimpleRectangle( 5, 5 );
16
17        System.out.println( "MAX WIDTH: " +
18            Utils.findMax( rects, new OrderRectByWidth( ) ) );
19    }
20 }
```



the functor

# FUNCTORS

- ▶ In the previous example call for `findMax`, the type of array whose maximum value should be found is `SimpleRectangle[]`
- ▶ Just by providing this array as first argument for `findMax`, Java figures out by itself that the instance of the type parameter `<AnyType>` of `findMax` must be `SimpleRectangle`
- ▶ Now, Java can verify that the method `compare` specified by the functor (that is, object `new OrderRectByWidth()`), is defined in a class (namely `OrderRectByWidth`) which implements interface `Comparator<SimpleRectangle>`, so we know we are using the right functor here. → Type safety!

# FUNCTORS

- ▶ Summing up, the `findMax`-example showed two things:
  - ▶ How to use functors to pass on methods (i.e., functions) into another method (here: a compare-method being passed as argument to `findMax`)
  - ▶ Also, how to do this in a type-safe way, by making everything generic

# FUNCTORS

- ▶ But again, functors can also be defined and used with non-generic interfaces, classes and methods.
- ▶ A functor is simply an object whose purpose is to encapsulate a method, in order to be able to pass around this method to somewhere else (e.g., as an argument of another method, or to store the method indirectly in a variable...)
- ▶ We used generics here to increase the type safety of `findMax` (and also to demonstrate another practical use case for generics)

# FUNCTORS

- ▶ Another example for the use of functors are callbacks in event handling, e.g., for -->GUIs. The user triggers some event (e.g., by clicking with the mouse or pressing a key on the keyboard). Some function should be called when this happens. The function is defined by the programmer and given as an argument to a built-in event manager:

functor (callback function)



```
eventManager.setCallBack(myMouseClickedHandler,  
MOUSECLICKEVENT);
```

- ▶ Each time the user clicks a mouse button, the `eventManager` method calls the method defined "inside" the functor `myMouseClickedHandler`
- ▶ `myMouseClickedHandler` is simply an object which has a certain method (the event handler)
- ▶ No generics involved here



# The Collections Framework

25

- ❑ Arrays are only efficient in case we require only random access (but no insertion or deletion) and the length of the collection does not require to increase or shrink.
- ❑ Furthermore, there are not many predefined operations for arrays.
- ❑ For other use cases, there are data structures which are more adequate, for example:
  - ❑ ArrayLists and linked lists
  - ❑ Stacks and queues
  - ❑ Maps
  - ❑ Java 8 Streams (not really a data structure but a way to access and compute data --> later)
- ❑ As you already know, dynamic data structures are provided by the *Java Collections Framework (Collections API)*...

# The Collections Framework

26

- Brief revision...
- A *collection* in this sense is an instance of one of the *Collections classes and interfaces* which are part of the Java Collections Framework a.k.a. Java Collections API.
- The Collections Framework forms a part of the Java API.
- Package: `java.util`

# The Collections Framework

27

- Each collection is a data structure
- A collection can be seen as a sort of *container*
- Stores a number of objects and allows for operations in order to modify, extend or read its content

# The Collections Framework

28

- ❑ Collections have a *dynamic size*.
- ❑ In contrast to arrays, they allow to add or delete objects in a way that the size of the collection is different after the operation.
- ❑ Collections can contain only objects as elements
- ❑ In order to store a primitive value in a collection, it needs to be boxed using a *wrapper class* (e.g., `int` needs to be boxed as `Integer`, `double` as `Double`...). Normally, Java does this for you automatically
- ❑ You already know some collection classes, e.g., `ArrayList`

# The Collections Framework

29

- Most collection classes and interfaces implement or are derived from interface `Collection`. This interface provides the basic operations which are allowed for most (but not all) kinds of collections:
  - ▣ `add(e)` : adds object `e` to the collection.
  - ▣ `remove(e)` : removes one object `e` from the collection.
  - ▣ `clear()` : removes all objects from the collection.
  - ▣ `contains(e)` : searches the collection. Results in `true` *iff* `e` is contained in the collection.
  - ▣ `size()` : results in the number of elements currently in the collection.
  - ▣ `equals()` : compares the collection content-wise with another collection.
  - ▣ `isEmpty()` : `true` *iff* collection is currently empty.
  - ▣ ...plus a few more methods (see Java API documentation)

# The Collections Framework

30

- There are various collection classes, each for a specific purpose. Each implements one of the following interfaces:
  - ▣ Set
  - ▣ List
  - ▣ Queue
  - ▣ Map
- From these interfaces stem various interfaces, abstract or concrete classes, such as for *sorted maps*, *stacks*, *vectors*, or *hash sets*.

# The Collections Framework

## Sets

31

- The `Set` interface extends interface `Collection`.
- However, it does not specify any additional methods, but it is used as a "marker interface" in order to mark a collection class as a set class.
- In contrast to an array or a list or a vector, a set does not contain any duplicate elements:

$$S = \{ e_1, e_2, \dots, e_n \}$$

$$S \cup \{ e_i, e_i \in S \} = S$$

- Furthermore, a set has no particular order:

$$\{ e_1, e_2, \dots, e_n \} = \{ e_n, e_1, e_2, \dots, e_{n-1} \} = \dots$$

However, Java also provides "ordered sets".

# The Collections Framework

## Sets

32

- Concrete classes which directly or indirectly implement **interface** `Set` **are** `HashSet`, `LinkedHashSet` **and** `TreeSet`.
- These are all sets, but differ in the way they organize their elements.
- For now, you can apply these classes without bothering how they are implemented internally.



# The Collections Framework

## Sets

33

- However, you need to know about the application areas and the efficiency of these classes:
  - ▣ `HashSet` is a `Set` implementation based on hashing. Adding, removing and looking up set elements happens in constant time (=very efficient). If you just need "a set", this class is the best choice.
  - ▣ `LinkedHashSet` maintains the order of the elements (that is, it is not a plain set in the mathematical sense).  
Elements are retrieved in the order in which they were inserted. But `LinkedHashSet` works often slower than `HashSet`.
  - ▣ `TreeSet` automatically sorts its elements w.r.t. the order imposed by the results of method `compare` of interface `Comparator` (so again, this is not a plain "mathematical" set)
  - ▣ See next slide for examples...

# The Collections Framework

## Sets

34

### □ Example (1):

```
import java.util.*;

...

Set s = new HashSet(); // creates an empty set
s.add(new Circle(0.5)); // lets add a few elements...
Circle c = new Circle(123.456);
s.add(c);
s.add(new Rectangle(1.2, 7.8));
s.remove(c); // removes an element
```

# The Collections Framework

## Sets

35

### A slightly more complex example:

```
Set sx = new LinkedHashSet(); // creates an empty "ordered set"
sx.add(new Rectangle(50,7)); // we add two items to the collection
sx.add(new Circle(0.5));
```

```
Object[] a1 = sx.toArray(); // we convert the set into an array
for(int k = 0; k<a1.length; k++)
    System.out.println(a1[k]);
```

```
Set sortedSet = new TreeSet(sx); /* transforms the LinkedHashSet
instance into a TreeSet instance */
```

# The Collections Framework

## Maps

36

How to use a TreeSet with a self-defined sorting order:

```
class MyClass { ... } // class of the items in the tree-set
class MyComparator implements Comparator<MyClass> { // a functor class
    public int compare(MyClass o1, MyClass o2) {
        return 1 or -1 or 0, depending on how o1 and o2 relate to
        each other according to the desired sorting order (see Lecture Section 4). Defines the
        sorting order of entries in the tree set
    }
}
```

```
TreeSet<MyClass> treeSet = new TreeSet<MyClass>(new MyComparator());
MyClass item1 = new MyClass(...);
treeSet.add(item1); // adds a new item to the tree-set
```

*// Note: If the items' class is a built-in class with a natural sorting order (which you want to use), such as Integer, then no comparator is required*

# The Collections Framework

## Generic classes

39

- Some of the previously shown examples for creation of instances of a collection class are quite simple, but have a serious shortcoming:

Without type arguments, they allow to store all sorts of objects mixed in the same collection, even wrong ones

=> we don't take advantage of type safety and from the fact that all collection classes/interfaces are generic!

# The Collections Framework

## Generic classes

40

- Therefore, we should always use collections together with a `<type argument>` (see lecture about generics):

```
Set<GeometricShape> s =  
    new HashSet<GeometricFigure>();  
s.add(new Circle(0.5)); // lets add a few elements..  
Circle c = new Circle(123.456);  
s.add(c);  
s.add(new Rectangle(1.2, 7.8));  
s.remove(c); // removes an element
```

- This way, it is not possible anymore to add a "wrong" object:

```
s.add(new Employee()); // fails at compile time!
```

# The Collections Framework

## Generic classes

43

□ How does Java implement, e.g., a generic `Set` ...?

□ Somewhat abbreviated:

```
public interface Set<E> extends Collection<E> {  
    public int size();  
    public boolean isEmpty();  
    public boolean add(E item);  
    public boolean contains(E item);  
    ...  
}
```

□ In `Set<GeometricShape>`, `GeometricShape` is the concrete type which replaces `E` in the interface declaration whenever an object of type `Set<GeometricShape>` is created.

# The Collections Framework

## Generic classes

44

- Again, the main benefit of generic classes and interfaces ("generics") is that they allow for a more rigid type checking, which avoids runtime errors.
- *Static type checking*: the compiler checks whether variables, arguments, expressions etc. have a static data type which is allowed at the respective place in your code.



# The Collections Framework

## Generic classes

45

- An example for a problem which would have been prevented if generics had been used (assuming that class `GeometricFigure` has an implementation of method `calcArea()`):

```
Set s = new HashSet();
s.add(new Circle(0.5));
...
s.add(new Employee());
...
Object[] array = s.toArray();
for(Object item: array)
    ((GeometricFigure)item).calcArea(); //fails at runtime!
```

# The Collections Framework

## Generic classes

46

- ❑ Static type checking is a significant benefit of Java over dynamically-typed programming languages (such as Python)
- ❑ In the context of types and type checking (other than in every-day language use) "dynamic" is bad, "static" is good
- ❑ Static typing makes Java suitable for large, complex software (although it can be used for small programs too, of course)
- ❑ Also static typing often increases the speed of Java programs (can you guess why?)

# The Collections Framework

## Lists

49

- Another important kind of collections: *lists*
- In contrast to sets, lists allow for duplicate elements and they always maintain a certain order among the items.
- Lists could alternatively be implemented using arrays, as we have seen, but the use of one of the Collections classes for lists instead might be much more powerful.

# The Collections Framework

## Lists

50

- **Interface `List`** specifies the basic operations (in addition to those declared by **interface `Collection`**). Note: the first element has index 0.
  - ▣ `add(index, e)` : inserts object `e` at position `index` into the list. In contrast to arrays, the current element at this position is not replaced, but the existing elements from position `index` on are shifted to the right to make room for the new item.

If `index` equals the current size of the list, `e` is appended to the list.
  - ▣ `get(index)` : retrieves the object which is stored at position `index`
  - ▣ `indexOf(e)` : yields the first index of element `e`
  - ▣ `lastIndexOf(e)` : yields the last index of element `e`
  - ▣ `remove(e)` : removes the element `e`. In contrast to an array, this does not leave a gap - the following elements are shifted left.
  - ▣ `set(index, e)` : replaces the element at position `index` with `e`
  - ▣ `subList(fromIndex, toIndex)` : yields a sublist

# The Collections Framework

## Lists

51

- **Classes** `ArrayList` and `LinkedList` implement interface `List`
- `ArrayList` stores the list elements similar to an array, that is, the elements are stored as a linear sequence in memory (one after the other). This allows for fast random access.
- But an `ArrayList` is - in contrast to an array - resizable (it can grow and might shrink) and you can insert new elements at any position.
- `ArrayList` is one of the most important and most frequently used collections in Java. It is often preferred over plain arrays.
- In contrast, a `LinkedList` stores the elements in a *linked list*, that is, a list where the elements are connected by pointers and do not necessarily form a continuous block in memory.

# The Collections Framework

## Lists

52

- ❑ Which array/list-like collection to use...?
- ❑ If you only need access and updates of existing elements use an array
- ❑ If your foremost requirement is speedy random access, but you also need to add new elements at the end of the list, `ArrayList` is typically the best choice
- ❑ If you frequently add new elements (or remove elements), at arbitrary positions (e.g., in the middle), use a `LinkedList`
- ❑ If you have no clue what to use, use an `ArrayList`

# The Collections Framework

## Lists

53

### □ Example:

```
ArrayList<Circle> cl = new ArrayList<Circle>();
```

```
cl.add(new Circle(0.0));
```

```
cl.add(new Circle(1.0));
```

```
cl.add(new Circle(2.0));
```

```
cl.add(1, new Circle(3.0));
```

```
cl.get(0).display();
```

```
cl.get(1).display();
```

```
cl.get(2).display();
```

```
cl.get(3).display();
```

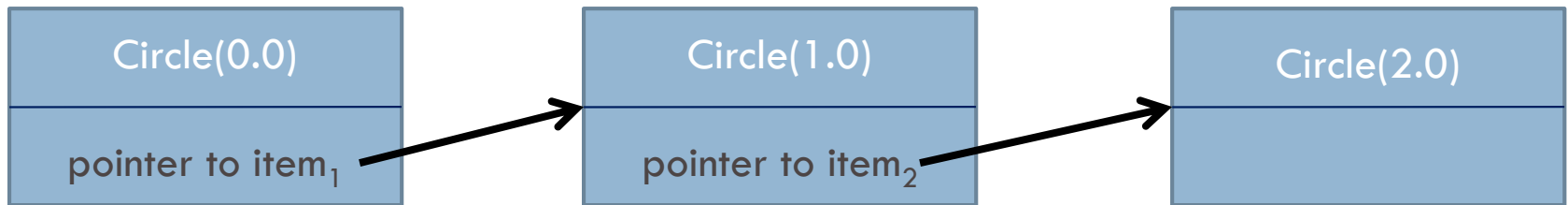
# The Collections Framework

## Lists

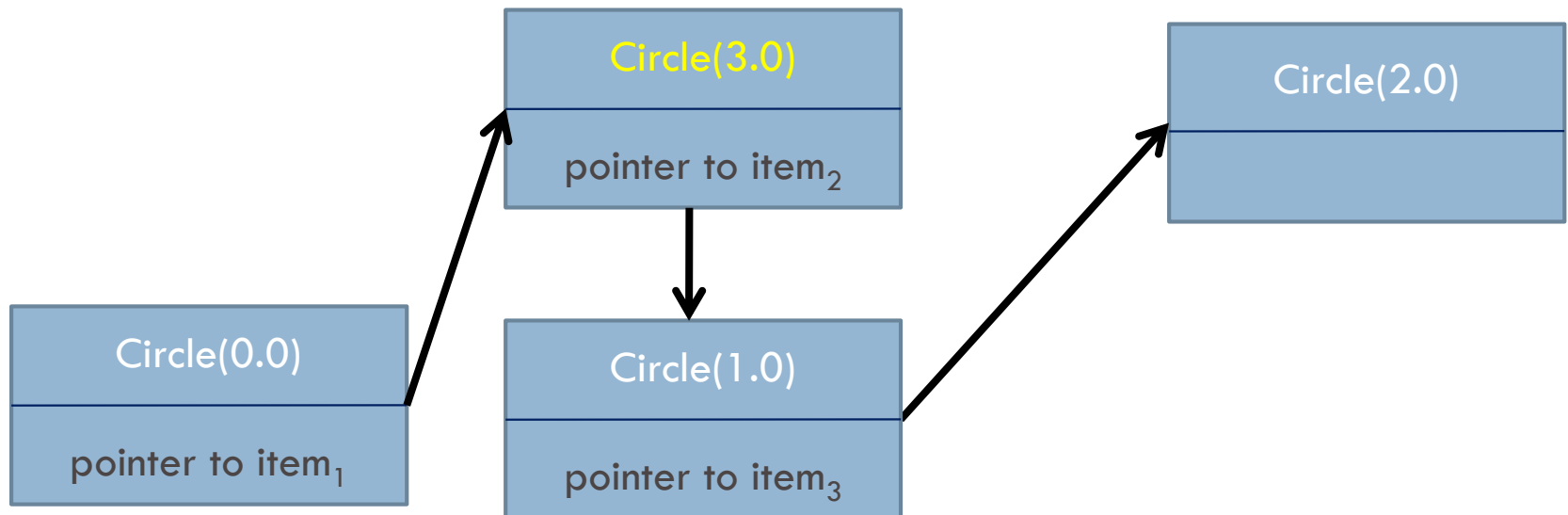
54

- Inserting a new item into a linked list:

Before:



After:



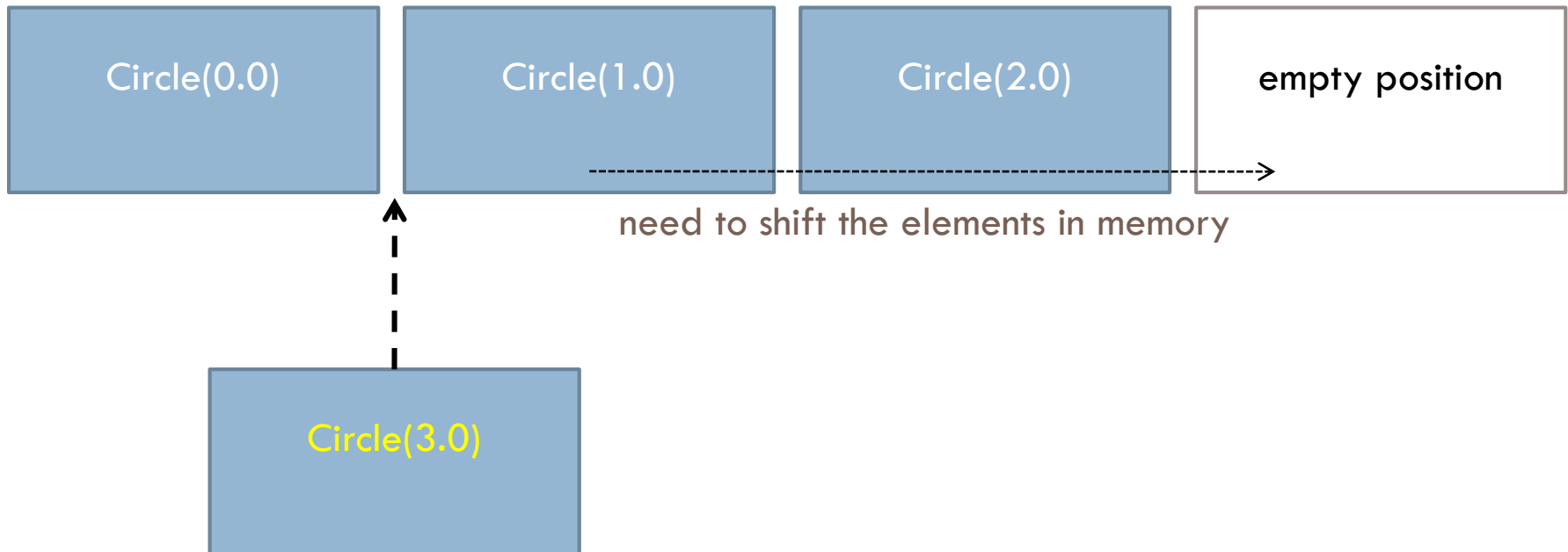


# The Collections Framework

## Lists

55

- ...compared to inserting a new item into an `ArrayList`:



# The Collections Framework

## Lists: Vector and Stack

56

- Further kinds of lists (implementing the `List` interface) provided by the Java Collections framework are `Vector` and `Stack`.
- `Vector` works much like `ArrayList`, but in contrast to an `ArrayList`, the internal operations of `Vector` are safe to be used *concurrently* (quasi-simultaneously) by two or more *threads*. But you'd still need to make your own operations which *use* the `Vector` thread-safe manually!
- On the other hand, this is slower...
- Concurrency will be a topic of one of the next lectures.

# The Collections Framework

## Lists: Vector and Stack

57

- `Stack` is derived from `Vector` and allows for *last-in first-out* access methods, that is, this kind of list is accessed at one end only (called the "top of the stack", although the access end could likewise be at the bottom - see coursework):
- `pop()` : returns and removes the top element from the stack.
- `push(e)` : adds a new element `e` at the top of the stack.
- `peek()` : returns the top element but does not remove it.
- `search(e)` : returns the position (index) of `e` in the stack.
- Remember to use all collections as generic classes, e.g.,  

```
Stack<Integer> myBirdStack = new Stack<Integer>();
```

# The Collections Framework

## Lists

60

- ...and there is also a class `Collections`, which is not a data structure type itself but a kind of "utility class"
- It provide several static methods for sorting, filling, reverting, copying, searching and other operations on lists.

```
ArrayList<Circle> cl = new ArrayList<Circle>();  
... // add elements  
Collections.sort(cl);
```

# The Collections Framework

## Maps

61

- Finally, the Collections framework provides classes and interfaces for *maps*
- Maps allow to associate search *keys* with values, in form of *key-value pairs*. We also say the map *maps* keys to values.
- The keys work like indices for retrieving specific values by their respective keys, but keys can be arbitrary objects, not just integer values like the indices of arrays
- Very important data structure! Similar structures also exist in other languages (e.g., Python's *dictionaries*)

# The Collections Framework

## Maps

62

- **Interface `Map`** specifies among other methods the following basic map methods:
  - ▣ `containsValue(v)`: returns true iff(\*) object `v` is used as a value in this map. Note that each value can have multiple keys.
  - ▣ `get(k)`: returns the value associated with key `k`. Quite fast - in particular much faster than searching for an arbitrary item in, e.g., an `ArrayList`
  - ▣ `put(k, v)`: puts a key-value pair `(k, v)` into the map.
  - ▣ `remove(k)`: removes the association (mapping) for key `k`.
- **For the complete set of operations, see the Java API documentation.**

(\*) "iff" means "if and only if"

# The Collections Framework

## Maps

63

- **Classes** `HashMap`, `LinkedHashMap` and `TreeMap` provide implementations of interface `Map`
- `HashMap` is the most basic of these classes. Use it if you don't require any specific ordering of the key-value pairs in the map.
- `LinkedHashMap` provides in addition an order of the key-value pairs, by linking them in the map much like with a linked list. This means that if the map is traversed, this happens in the specified order (and not in some unspecific sequence).
- `TreeMap` in addition maintains a specified *sorting* order among the keys - see examples on later slides here


# The Collections Framework

## Maps

64

Type of the keys

Type of the values



```
HashMap<String, String> hama =  
    new HashMap<String, String>(); // a new empty hash map
```

```
//We add a few key-value pairs to our hash map...
```

```
hama.put("a", "value 1");
```

```
hama.put("c", "value 3");
```

```
hama.put("b", "value 2");
```

```
//We retrieve and print the value associated with key "b":
```

```
System.out.println(hama.get("b"));
```

- Note that keys and values are not restricted to strings - you can use any kinds of objects, including instances of your own classes



# The Collections Framework

## Maps

65

How to use a TreeMap with user-specified key sorting order:

```
class KeyComparator implements Comparator<KeyClass> {  
    public int compare(KeyClass key1, KeyClass key2) {  
        return 1 or -1 or 0, depending on how key1 and key2 relate to  
each other according to the desired sorting order (see Lecture Section 4). Defines the  
sorting order of the (key,value) pairs in the tree-map  
    }  
}  
  
class KeyClass { ... } // class of the keys in the tree-map  
class ValueClass { ... } // class of the values in the tree-map  
  
TreeMap<KeyClass, ValueClass> treeMap =  
    new TreeMap<KeyClass, ValueClass>(new KeyComparator());  
  
KeyClass k1 = new KeyClass(...);  
ValueClass v1 = new ValueClass(...);  
  
treeMap.put(k1, v1); // adds a new (key,value) pair to the tree-map
```

*// Remark: If the keys' class is a built-in class with a "natural" sorting order (which you want to use), such as Integer, and you want to use this default order, then no comparator is required*

# The Collections Framework

## Iterators

66

- What if we want to *traverse* the elements of a collection (i.e., an object of a set / list / queue / map class)?
- Some of the Collections classes make it quite obvious how to do this: e.g., to traverse a stack, we could subsequently pop all its elements one by one (side effect: afterwards the stack is empty).
- But how to traverse, e.g., a `HashSet`...?
- *Iterators* provide a uniform way to traverse almost all kinds of collections

# The Collections Framework

## Iterators

67

- **Interface** `Collection` (which most `Collections` classes implement) provides a method `iterator`, which returns an *iterator* - an object of a class which implements the generic interface `Iterator`.
- The iterator can then easily be used to traverse the collection using its `hasNext` and `next` methods:

```
LinkedList<GeometricFigure> myList =  
    new LinkedList<GeometricFigure>();  
...  
Iterator<GeometricFigure> iterator = myList.iterator();  
while (iterator.hasNext())  
    iterator.next().display();
```

# The Collections Framework

## for-each with collections

68

- Furthermore, we can use a variant of the for-each loop to traverse collections, analogously to for-each in the context of arrays:

```
LinkedList<GeometricFigure> myList =  
    new LinkedList<GeometricFigure>();
```

```
...
```

```
for (GeometricFigure element: myList)  
    element.display();
```

# The Collections Framework

## for-each with collections

69

- ❑ Maps don't implement interface Collection. How to traverse a map...?
- ❑ You can traverse either the map's set of keys, or the map's set of values, or the set of (key, value)-pairs using a loop:

```
for(Map.Entry<String, Double> entry : treeMap.entrySet()) {  
    String k = entry.getKey();  
    Double v = entry.getValue();  
    System.out.println("key: " + k);  
    System.out.println("value: " + v);  
}
```

# The Collections Framework

## for-each with collections

70

- Another approach is to use an iterator over the set of values or keys. E.g., to traverse all `TreeMap` values in the sorting order of the map's keys:

Type of the values in the map



```
Collection<ValueClass> values = treeMap.values();  
Iterator<ValueClass> itr = values.iterator();  
while(itr.hasNext())  
    System.out.println(itr.next());
```