ADP

Data

HAN_UNIVERSITY OF APPLIED SCIENCES

Topics

1. Abstract Data Types
2. Lists
3. Stack
4. Queue
5. Generic Programming (& Wildcard)
6. Comparable

Abstract Data Types (ADT)

An **Abstract Data Type** (ADT) is a high-level **mathematical model** for data structures that defines a set of operations and the behavior of those operations without specifying the actual implementation details.

ADT - Before

Before the formal concept of Abstract Data Types (ADTs) emerged in computer science, programmers typically worked with data structures and algorithms in a less abstract and structured manner.

Here's what was prevalent before the explicit use of ADTs:

1. **Primitive Data Types** (integers, floating-point, characters)
2. **Custom Data Structures**

ADT - Examples

Common examples of ADTs include:

1. **Stack**: An ADT that supports two main operations, "push" to add an element to the top of the stack and "pop" to remove an element from the top of the stack.
2. **Queue**: An ADT that supports "enqueue" to add an element to the back of the queue and "dequeue" to remove an element from the front of the queue.
3. **List**: An ADT that allows for adding, removing, and accessing elements in a linear sequence.

ADT - Examples

4. **Dictionary or Map**: An ADT that stores key-value pairs and supports operations like inserting, retrieving, and deleting values based on their keys.
5. **Set**: An ADT that represents a collection of unique elements and supports operations like adding, removing, and testing for membership.

ADT – Data Structures

The actual implementation of these ADTs can vary, and different programming languages may provide built-in support for some of these abstract data types through **data structures**.

ADT – Components

An ADT typically consists of 2 main **components**:

1. **Data**: This is the underlying information that the ADT represents. It can be any type of data, such as integers, strings, or more complex data structures.
2. **Operations**: These are the functions or methods that can be performed on the data. Operations define how you can interact with the data, including creating, modifying, and accessing it.

ADT – Key concepts

The **key idea** is that the user of the ADT only needs to know what operations are available and what they do, without needing to know how these operations are implemented.

This separation of concerns allows for modularity and abstraction in software design, making it easier to manage and maintain code.

ADT – Key concepts

By defining ADTs, software developers can create reusable and understandable components in their programs and hide the internal details of data structures and algorithms, promoting good software engineering practices like encapsulation and information hiding.

ADT – Support for ADT

The emergence of high-level programming languages and concepts like **object-oriented programming** (OOP) further facilitated the adoption of ADTs, providing language constructs and libraries that supported abstract data types.
Allowed developers to define classes as abstract data types, which could be instantiated and used in a more structured and abstract manner.

OpenDSA - Abstract Data Types

Waarom je ADT’s en datastructuren leert

Even if you don't implement algorithms yourself, but only call them, you still have to work with their input and output data (often in data structures). It is then important to know what kind of operations to look for in the documentation.

In practice, the <u>implementation details</u> in data structures will have <u>(major) consequences for the performance efficiency</u> of an algorithm implementation or software system as a whole. That is why it is important to distinguish between the concepts of ADT and data structure.

Lists, stacks, and queues are the most important ADTs (except for the simple/primitive types). These mainly help you to run down items 'in a row' (sequential), also known as linear. You can learn more about this in the following.

Arrays (fundamental data structure - not ADT)

Advantage:

- Elements are directly addressable

Disadvantages:

- Fixed size
- Duration to insert/remove elements in the middle (due to sliding).
- Generics are tricky (making generic array).

Depending on the implementation, Lists solve one or more of these problems!

Wat zijn Lists? (ADT)

- A collection where the order and position matter.
- Examples: list files, drop-down list, to-do list, etc.

Belangrijkste operaties:

`add()` – **Place item at the back**

`get()`/`set()` – **Find/place item in a specific place**

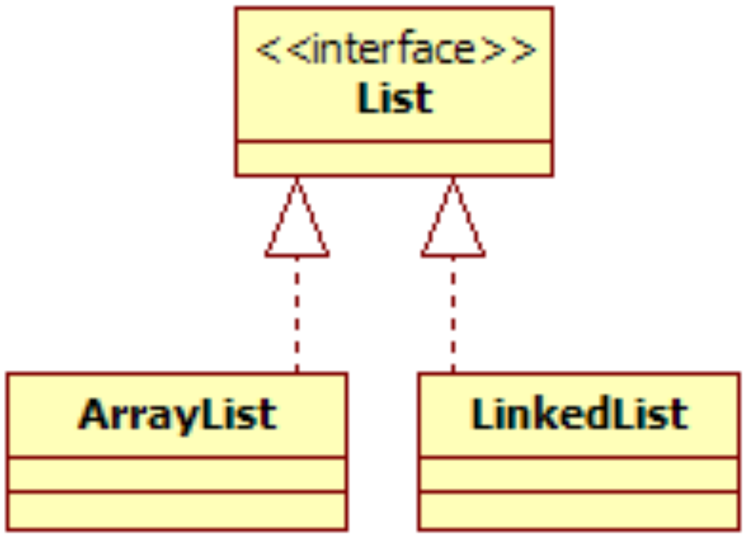
`remove()` – **Delete item**

`find()`/`contains()` – **See if there is an item in the list**

Implementatie list

A list can be used in Java via a:

- `ArrayList`
- `LinkedList`



Gebruik `ArrayList`

```java
// Gebruik generic ArrayList
ArrayList<String> listOfStrings = new ArrayList<String>();
listOfStrings.add("Michel");
String m1 = listOfStrings.get(0);
```

Beter: program to an interface, not an implementation

```java
// Gebruik generic ArrayList
List<String> listOfStrings = new ArrayList<String>();
listOfStrings.add("Michel");
String m1 = listOfStrings.get(0);
```

`ArrayList`

- `ArrayList` uses an ordinary array underwater:

3	6	8	0	5	6				

- `add()` **adds at the end.**
- **What needs to be done to place an item in front of or between two elements?**

Als de array volloopt...

- Create a new array, 2n+1 in size.
- Copy the old one into the new array.
- Code:

```
if (theItems.length == theSize) {
    AnyType[] old = theItems;
    theItems = (AnyType[]) new Object[theSize*2+1];
    for (int i=0; i<theSize; i++) {
        theItems[i] = old[i];
    }
         theSize = theSize*2 + 1;
}
```

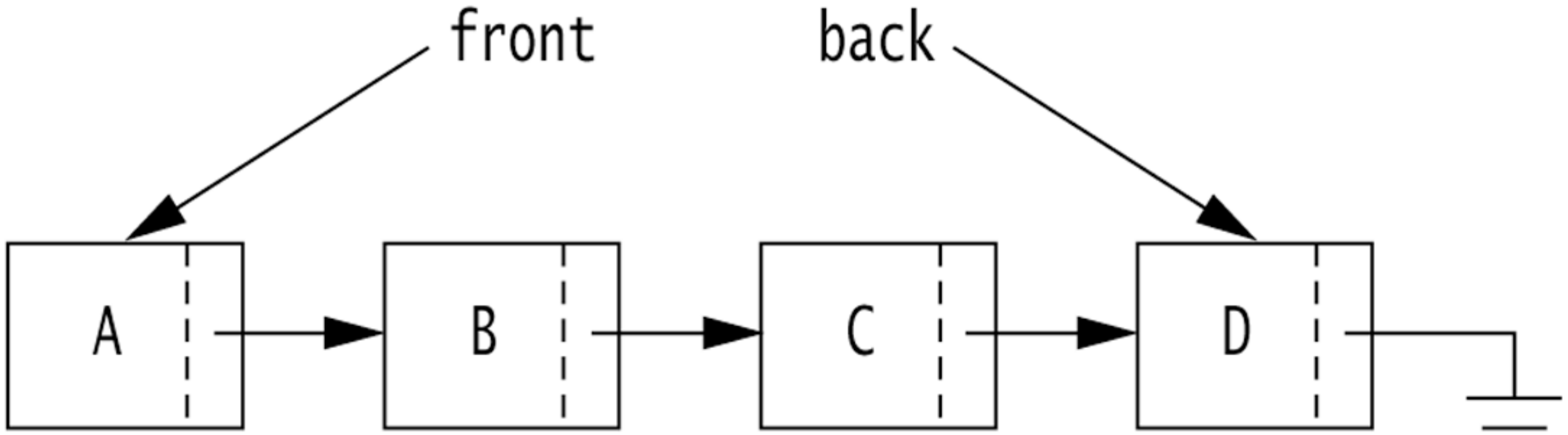
Gebruik `LinkedList`

```
// Gebruik generic LinkedList
LinkedList<String> listOfStrings = new LinkedList<String>();
listOfStrings.add("Michel");
String m1 = listOfStrings.get(0);
```

Beter: program to an interface, not an implementation

```
// Gebruik generic ArrayList
List<String> listOfStrings = new LinkedList<String>();
listOfStrings.add("Michel");
String m1 = listOfStrings.get(0);
```

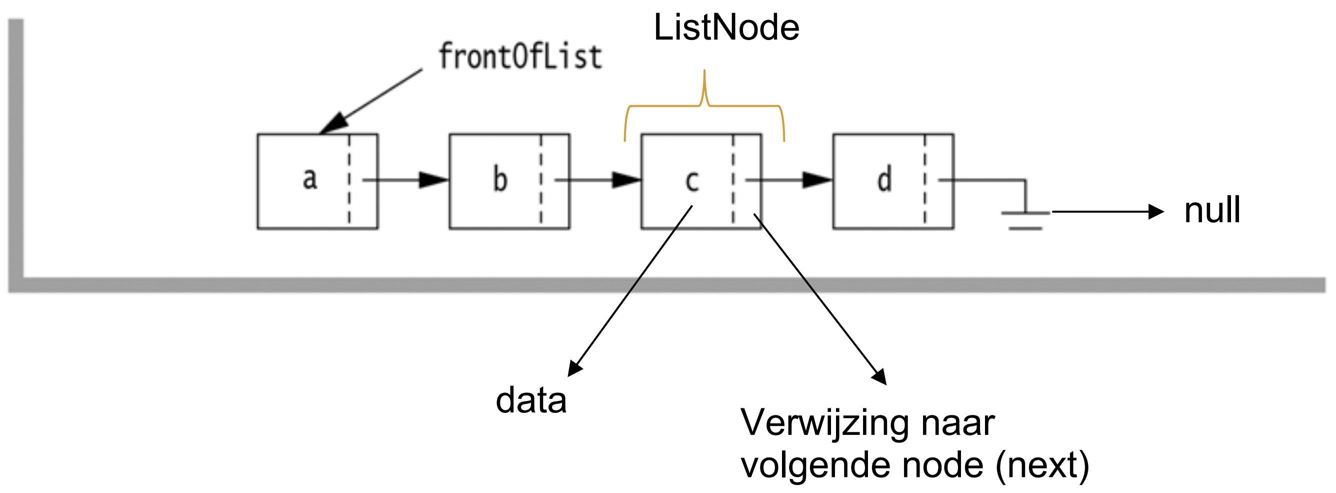
Linked list implementation



Singly Linked List

figure 17.1

Basic linked list



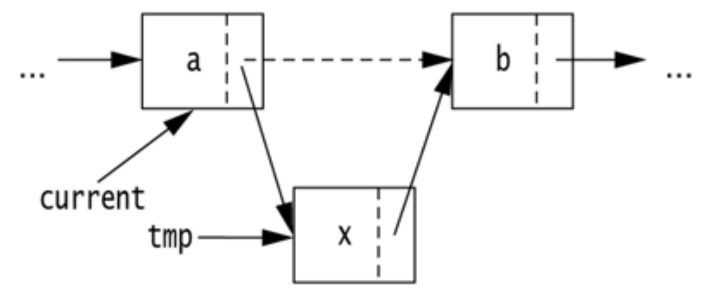
```
public class ListNode {
    Object element;
    ListNode next;
}
```

```
public class ListNode<T> {
    T element;
    ListNode<T> next;
}
```

Insertion in Linked List

figure 17.2

Insertion in a linked list: Create new node (`tmp`), copy in `x`, set `tmp`'s next link, and set `current`'s next link.



Deletion in Linked List

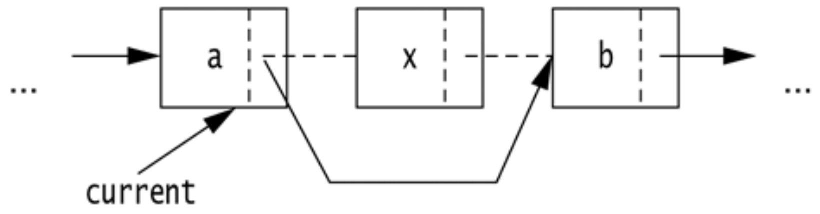


figure 17.3

Deletion from a linked list

Header Node

figure 17.4

Using a header node for the linked list

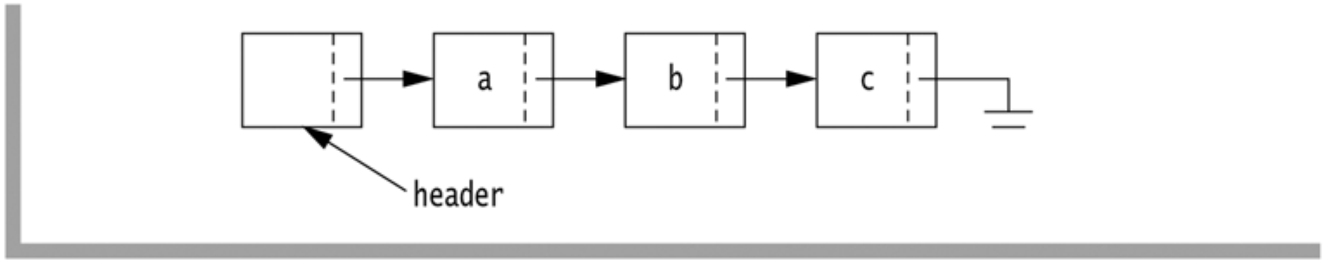
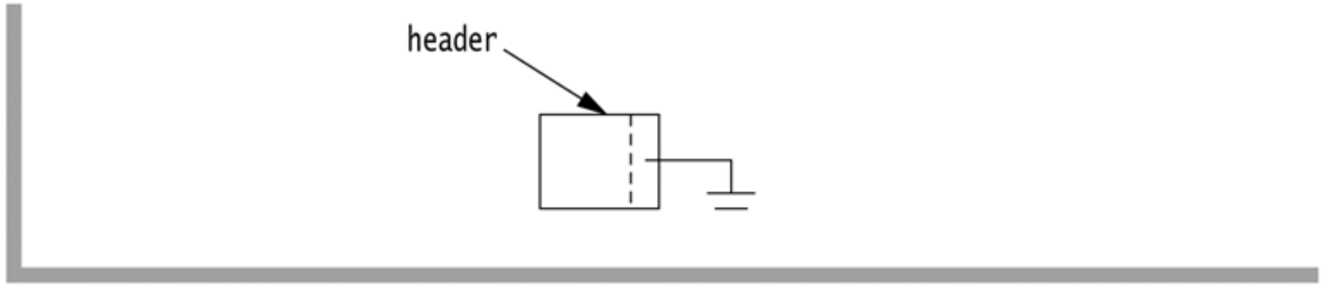


figure 17.5

Empty list when a header node is used



- **Always one node present.**
- **Resolves development issues.**

Doubly Linked List

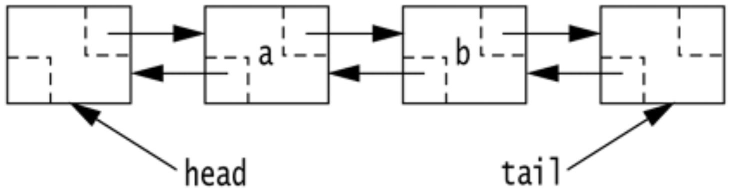


figure 17.15

A doubly linked list

`LinkedList` versus `ArrayList`

- Adding elements: `LinkedList` typically better than `ArrayList`.

- Accessing any element "at once": `ArrayList` typically better than `LinkedList`.

Lijst (list) – ADT (extra)

The list is the ADT for a finite, ordered sequence of items/elements that don't have to be unique.

In the following videos, David Scot Taylor explains various ADTs and data structures in an accessible way.
Lists, Iterators and Abstract Data Types.

[OpenDSA - Lists]
[Itereren (op Ans)]

[Video Link]

Playlist

At least:
- Implementations
  - Linked Lists
  - Array Lists
- Comparison (use and misuse)

David Taylor: Algorithms with Attitude, 2015

Waarom je lists leert (extra)

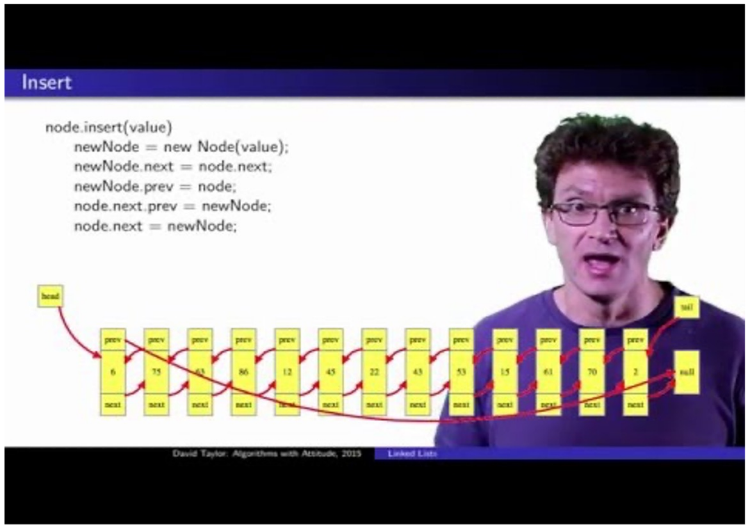
Every programmer has worked with a list at some point. Even an ordinary array from Java is essentially a data structure for a list. That array implementation meets the definition of list, but some operations (e.g., add an element at the beginning or end of it) are not implemented in a single function (e.g., method) by default.

De gelinkte lijst/linked list – ADT (extra)

A **linked list** is a **data structure** for a list, consisting of nodes. A node consists of the element (the content) and references to the next and sometimes previous node. In the latter case, it's called a **double-linked list**. The elements are not necessarily sequentially connected in the memory (i.e. not linear in the memory, but sequentially descended). The references are **pointers** or **references** (programming language-dependent) to a memory object.

OpenDSA - Linked Lists
OpenDSA - Doubly Linked Lists

Gelinkte lijsten en geheugenruimte (op Ans)
Sentinel node (op Ans)



```
node.insert(value)
    newNode = new Node(value);
    newNode.next = node.next;
    newNode.prev = node;
    node.next.prev = newNode;
    node.next = newNode;
```

Video Link

Waarom je gelinkte lijsten leert (extra)

Memory Management: Linked lists are used in memory management to keep track of available memory blocks. When you allocate memory dynamically (e.g., with malloc in C), the memory manager may use a linked list to find and allocate the appropriate memory block.

Web Browsers: Web browsers use linked lists to implement the backward and forward navigation history. Each visited web page is a node in a linked list, and you can traverse the history using backward and forward pointers.

Undo/Redo Functionality: Software applications often implement undo and redo functionality using linked lists. Each change in the application state can be represented as a node, allowing users to navigate backward and forward in the history of actions.

Linked lists are a possible underlying data structure for other data structures, such as stacks, queues, hashtables, and graphs.

Linked lists are used more often than alternative lists if the number of elements is not known or should not be fixed, and random access to elements is not necessary (e.g. not used for sorting).

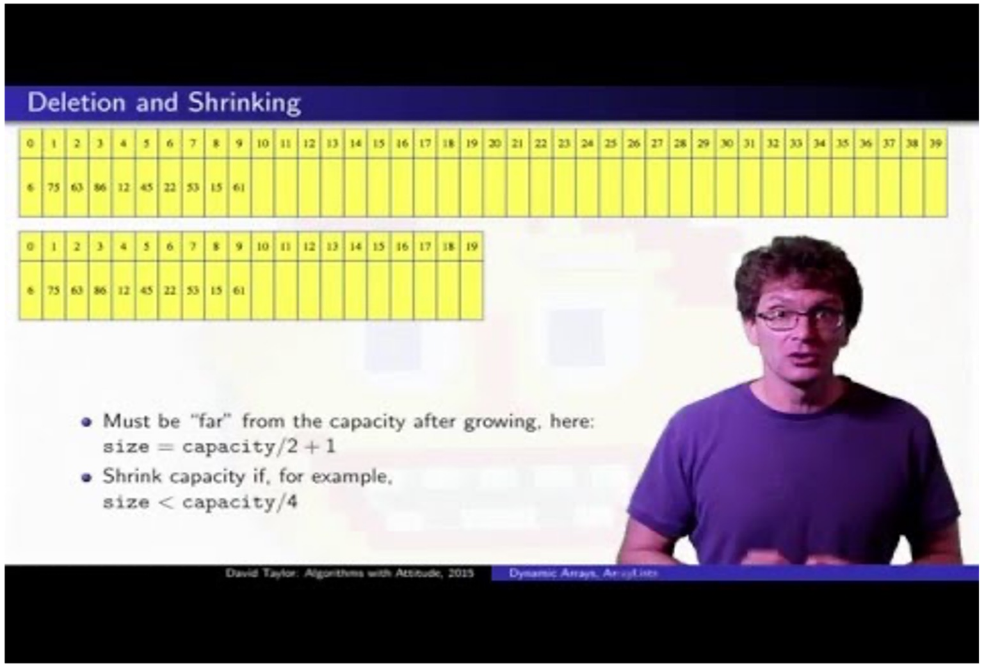
De dynamische rij/dynamic array (extra)

Dynamic rows or dynamic arrays are also known as vectors. Dynamic rows do not have a fixed length, static (regular) arrays do. This makes them very versatile, and perhaps the most widely used data structure ever.

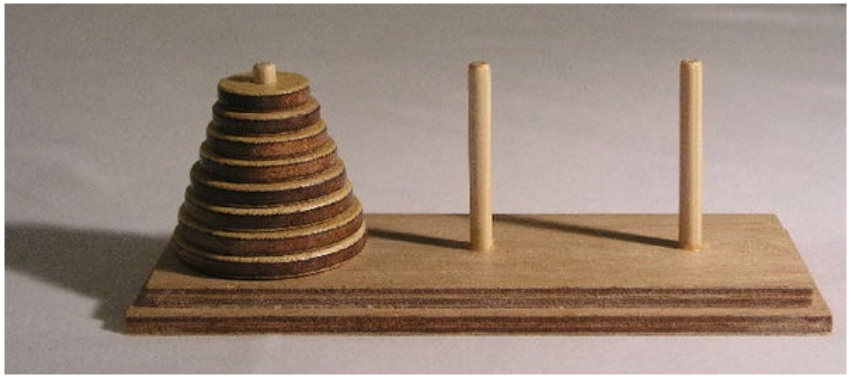
In the following videos, David Scot Taylor explains dynamic rows. By the way, you don't have to learn his explanation of amortized algorithm analysis for ADP.

[Array access (op Ans)]()
[ArrayList versus statische rij (op Ans)]()

[Video Link]()


Deletion and Shrinking

- Must be "far" from the capacity after growing, here: size = capacity/2 + 1
- Shrink capacity if, for example, size < capacity/4



STACK - ADT

Wat is een stack? (ADT)

- Stack = Stapel. Data structure where only the most recently added element is available
- LIFO: Last In First Out
- Basic operations:
  - `push()`
  - `pop()`
  - `top()`
  - `isEmpty()`

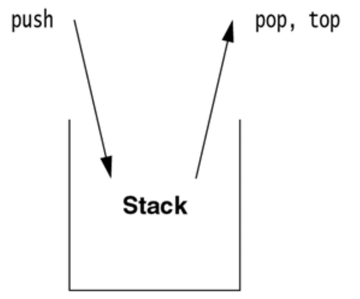
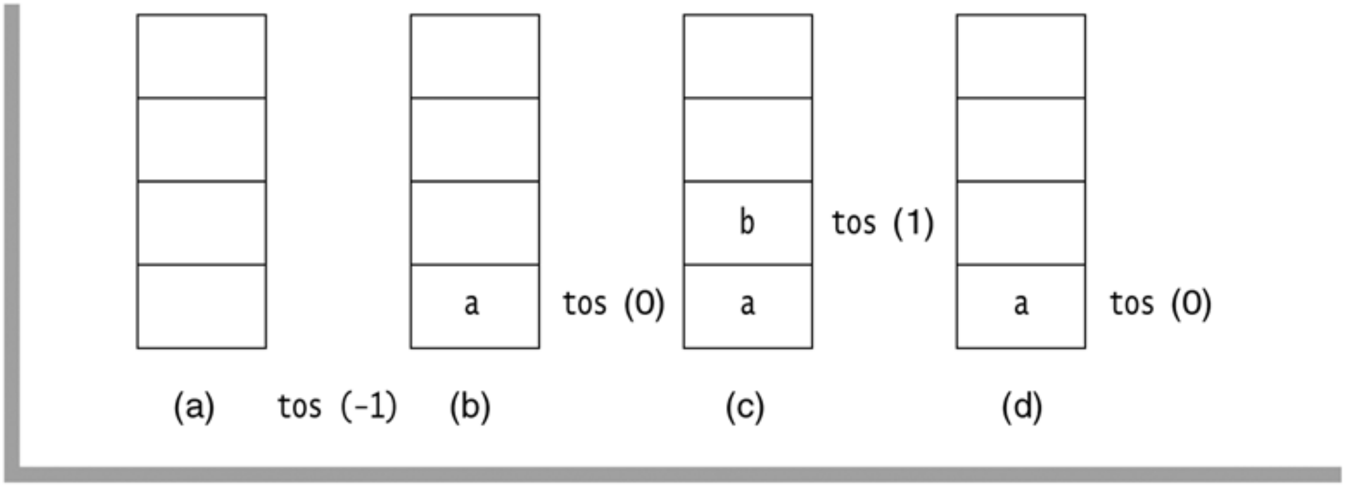


figure 6.20
The stack model: Input to a stack is by `push`, output is by `top`, and deletion is by `pop`.

Voorbeeld Stack

figure 16.1

How the stack routines work:
(a) empty stack;
(b) `push(a);`
(c) `push(b);`
(d) `pop( )`



Toepassingen

- Function calls (al in OOPD/SPD)
- (String) Reversion.
- In various algorithms.

APP: Algoritmen en datastructuren

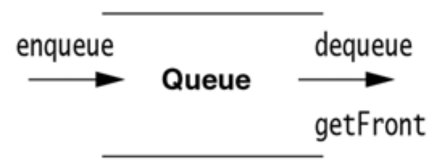
QUEUE

Wat is een queue? (ADT)

- A data structure in which only the first added element is available.
- New elements come to the back of the row.
  - FIFO: First In First Out
- Operations:
  - `enqueue()`
  - `dequeue()`
  - `getFront()`

figure 6.22

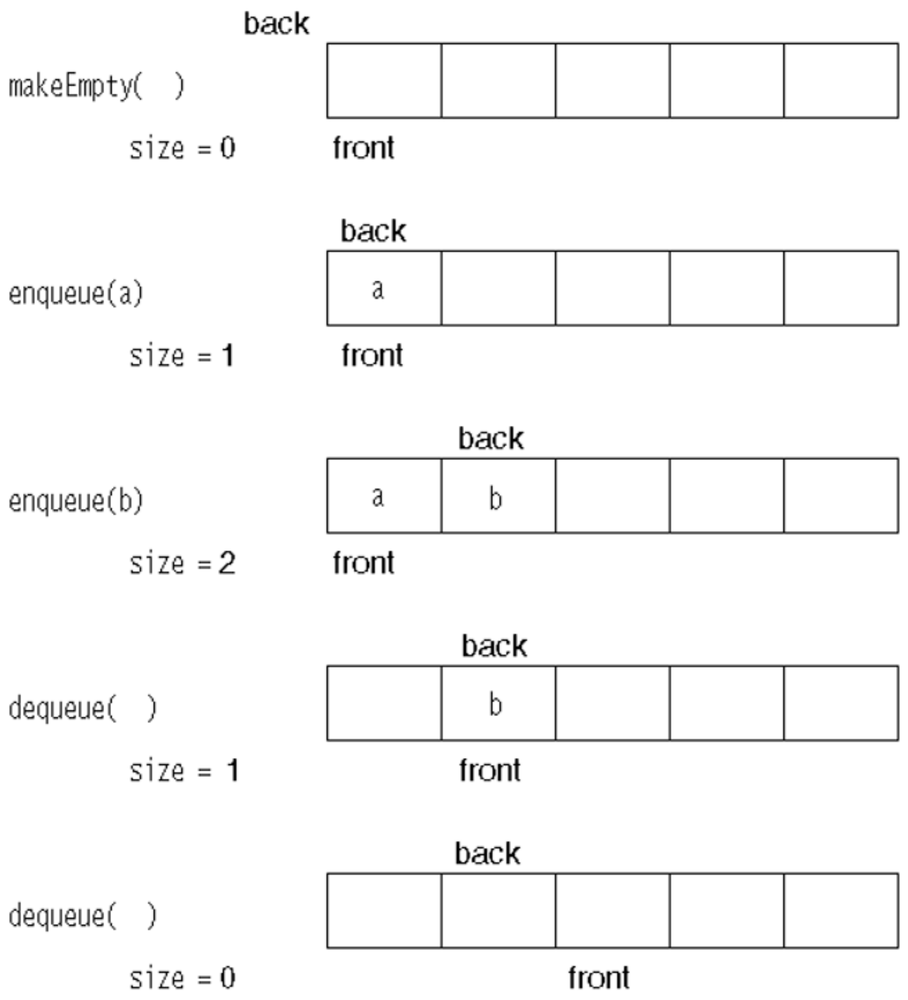
The queue model: Input is by **enqueue**, output is by **getFront**, and deletion is by **dequeue**.



Toepassingen

- Queues (printer queues, message queues).
- Buffers.

Array implementation van queue



So no shifts!

Priority Queue

- A queue that returns the lowest value every time(priority)
  - E.g. For prioritizing processes and/or threads
  - Operations `insert(),findMin(),deleteMin()`
- Needed later in this course (shortest-path algorithms)
- Of gebruik `java.util.PriorityQueue`
  - Operations: `add(),peek(),poll()`
    ipv. `insert(), findMin(),deleteMin()`

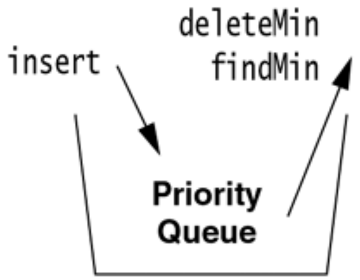


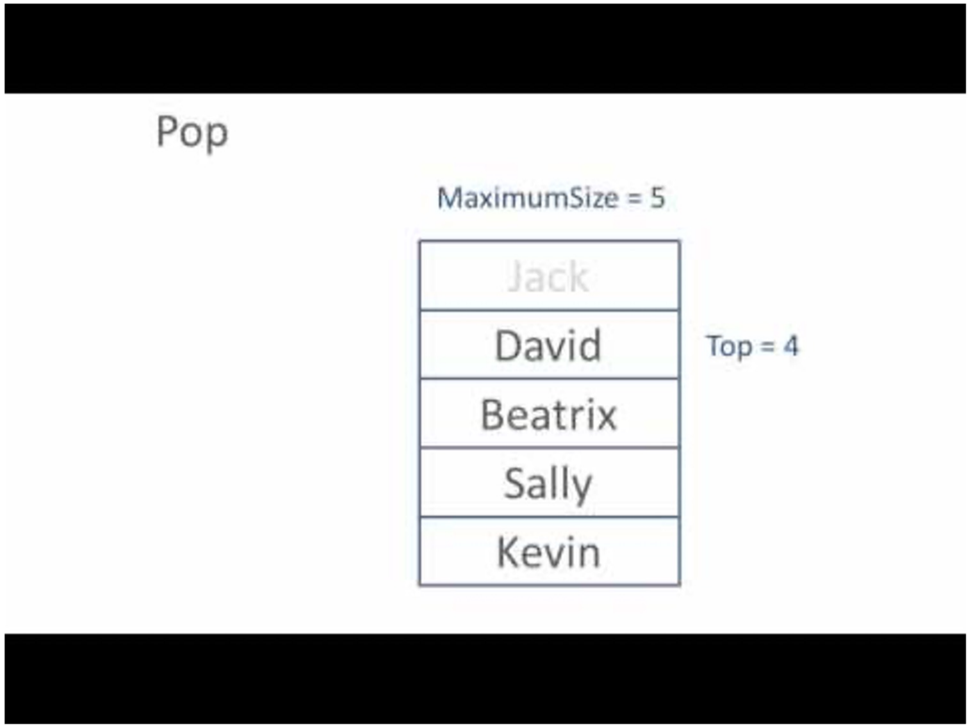
figure 6.34

The priority queue model: Only the minimum element is accessible.

De stapel/stack (extra)

The ADT stack returns the last inserted element first (last in, first out (lifo)).

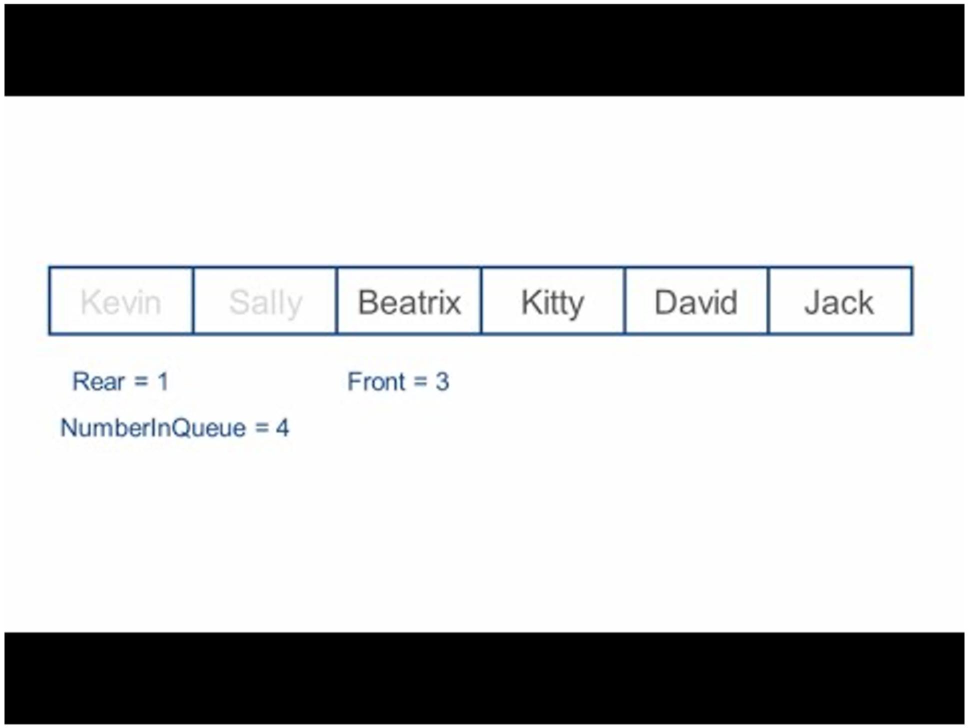
Open DSA: Stacks



Video Link

De wachtrij/queue (extra)

The ADT **queue** returns the earliest inserted element first (first in, first out (fifo)).



Extern: Queues

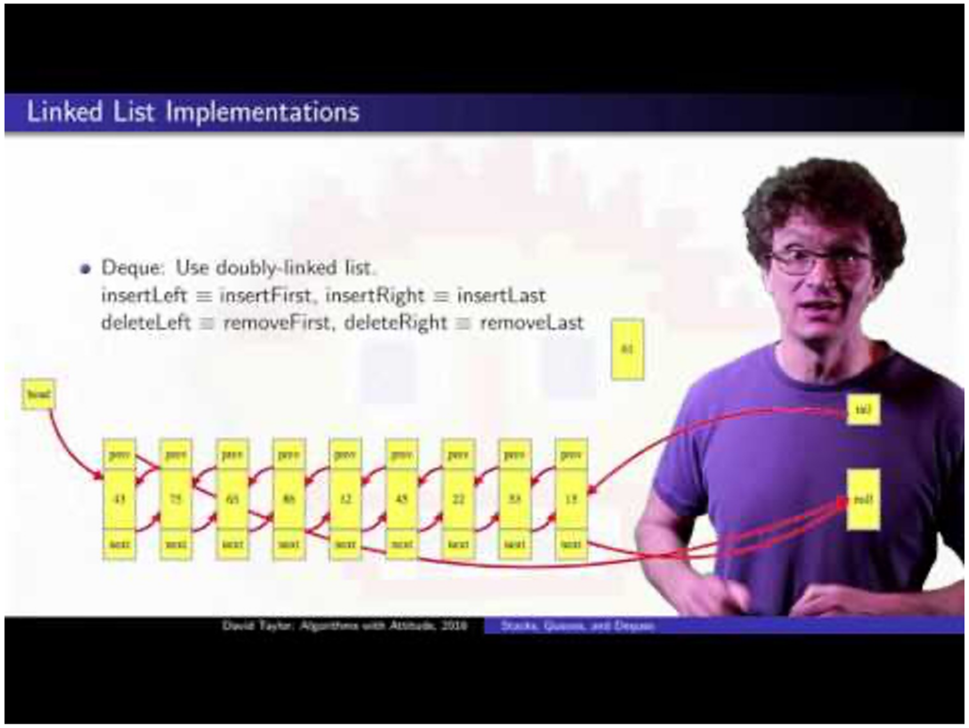
Video Link

De prioriteitswachtrij/priority queue (extra)

The ADT priority queue is identical in interface to a queue, but the operations have a different meaning. Instead of the earliest inserted element being returned first, the smallest element is returned each time. Alternatively, the largest element can be returned each time. More generally, the element with the highest priority is returned each time.

Van stacks en queues naar de deque (extra)

David Scot Taylor explains the connection between stacks, queues, and an even more generic ADT: deques.



Linked List Implementations

- Deque: Use doubly-linked list.
  insertLeft ≡ insertFirst, insertRight ≡ insertLast
  deleteLeft ≡ removeFirst, deleteRight ≡ removeLast

head
tail
null
David Taylor: Algorithms with Attitude, 2016
Stacks, Queues, and Deques

Video Link

GENERICS

Generic programming

Despite being more specific than **ADTs**, it is optimal to implement **data structures** generically in terms of the data that fits in them, **instead of writing out source code for all kinds of different conceivable types**, with a lot of repetition. Java is one of the programming languages that offers possibilities for **generic programming**. Generic is the counterpart of specific. Generic programming means that within a programming language that supports it, you can program rules about which data types are valid at a place in the source code where types are located. Generic, because those rules are more flexible and general than 'the class is exactly ThisClass'. With Java, you can apply generic programming in definitions of a method, class, or interface.

Oracle: Generics

Generic programming is useful to make source code more general, i.e. to make source code more reusable. Consider, for example, the following applications:

If you don't want to cast to a superclass, such as Object, but also can't use an exact data type.

If a data structure contains only elements of the same data type, your source code can be simplified and your object code can deliver higher performance efficiency.

Bekijk de volgende code

```
public class MemoryCell {
    private Object storedValue;

    public Object read() {
        return storedValue;
    }

    public void write (Object x) {
        storedValue = x;
    }
}
```

- What types can `MemoryCell` contain?

Gebruik MemoryCell

```java
public class MemoryCell {

    private Object storedValue;

    public Object read() {
        return storedValue;
    }

    public void write (Object x) {
        storedValue = x;
    }

    public static void main(String[] args) {
        MemoryCell m = new MemoryCell();
        m.write("37"); // Onthoud een string
        m.write(new File("readme.txt")); // Onthoud een File
        m.write(25); //Error! (Voor java 1.5)
    }
}
```

Wrapper Types

```
Object o = 25; // Mag niet (voor java 1.5)
                  want 25 is geen object!
```

- 25 is a `int`, a primitive type
- We "pack"” 25 in the wrapper type Integer and then it is allowed:

```
Integer i = new Integer(25); // Boxing
Object o = i; // Mag!
int j = i.intValue(); // Unboxing
```

- **For every primitive type there is a Wrapper Type.**
- **(Un) Boxing didn't come naturally before version Java 1.5.**
- **Boxing/Unboxing is expensive, be careful. Why?**
- **Since Java 1.5 autoboxing/unboxing.**

Autoboxing and Unboxing

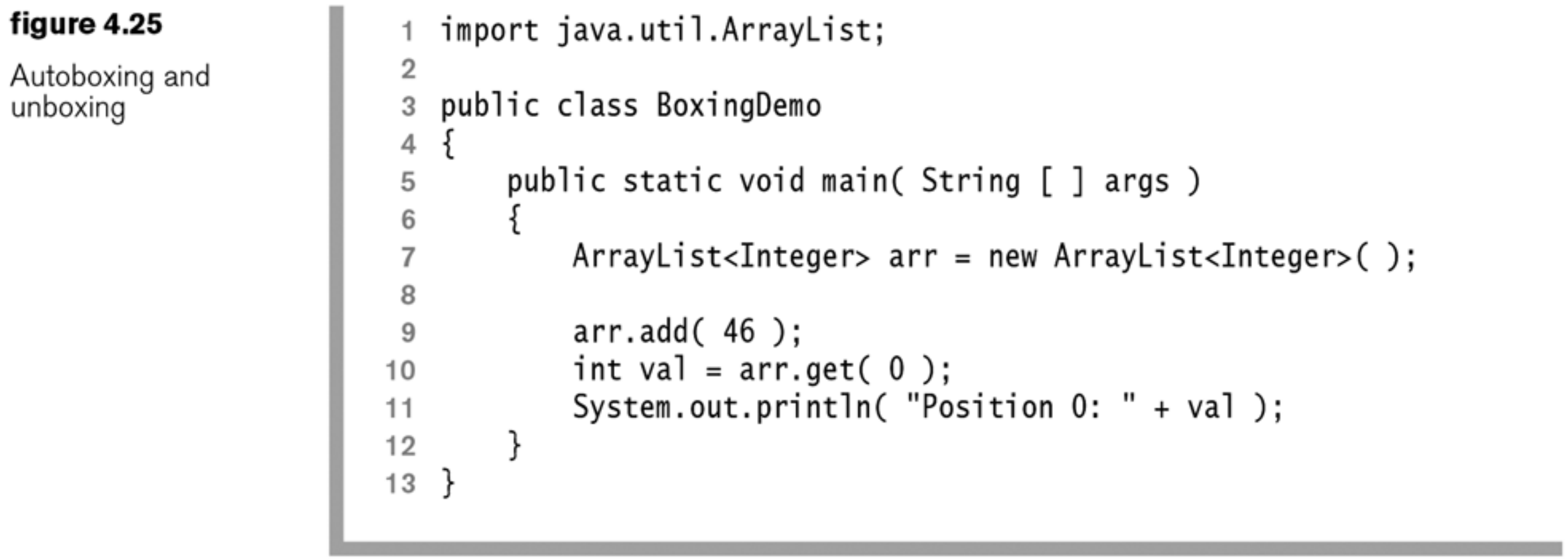


figure 4.25

Autoboxing and unboxing

```
import java.util.ArrayList;

public class BoxingDemo
{
    public static void main( String [ ] args )
    {
        ArrayList<Integer> arr = new ArrayList<Integer>( );

        arr.add( 46 );
        int val = arr.get( 0 );
        System.out.println( "Position 0: " + val );
    }
}
```

- **Vanaf Java 1.5.**

MemoryCell revisited

```
public class MemoryCell {
    private Object storedValue;

    public Object read() {
        return storedValue;
    }

    public void write (Object x) {
        storedValue = x;
    }
}
```

- `read`() gives an `Object` back, but what type is it exactly? You don’t know!
- Question: Can we `MemoryCells` for a specific type?

Generic class MemoryCell

```
public class MemoryCell<T> {
    private T storedValue;

    public T read() {
        return storedValue;
    }

    public void write (T x) {
        storedValue = x;
    }
}
```

- `T` (self-chosen name!) stands for parameter type.
- In the event of instantiation of `MemoryCell` you have to indicate which type `MemoryCell` may contain.
  `MemoryCell<Student> cell;`
- We say: `MemoryCell` is generic for types `T`.

Generic class MemoryCell: usage

```java
public class MemoryCell<T> {
    private T storedValue;

    public T read() {
        return storedValue;
    }

    public void write (T x) {
        storedValue = x;
    }

    public static void main(String[] args) {
        // Maak een MemoryCell voor alleen Integers:
        MemoryCell<Integer> m = new MemoryCell<Integer>();
        m.write(25); // Auto boxing!
        m.write("25"); // Fout!
    }
}
```

Generic method

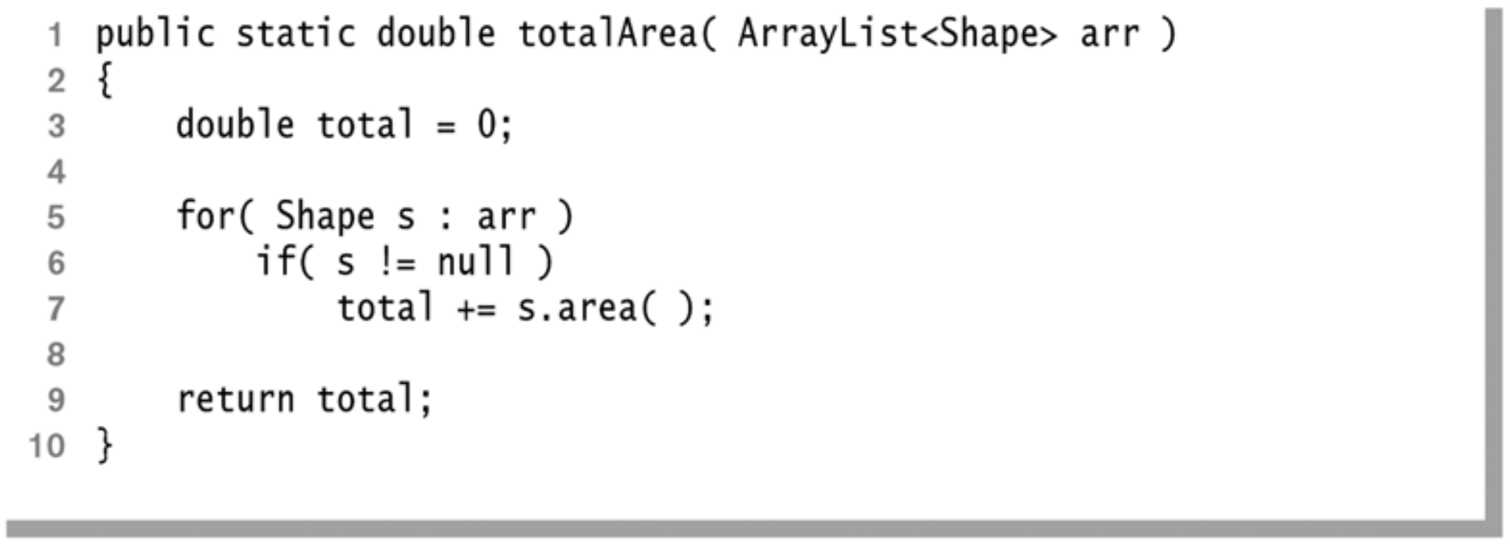
```
public static <AnyType>
boolean contains( AnyType [ ] arr, AnyType x )
{
    for( AnyType val : arr )
        if( x.equals( val ) )
            return true;

    return false;
}
```

Voordeel van Generics

- You already know during compile-time which type is/can be used.
- Errors are therefore noticed during compiling, and not only at runtime.

Probleem

- Let's say we have a class `Square` which is inherited from a class `Shape`.
- The code below does not work if you pass a `ArrayList<Square>`
- This is because the `ArrayList<Square>` not inherited from `ArrayList<Shape>`



```java
public static double totalArea( ArrayList<Shape> arr )
{
    double total = 0;

    for( Shape s : arr )
        if( s != null )
            total += s.area( );

    return total;
}
```

figure 4.30

`totalArea` method that does not work if passed an `ArrayList<Square>`

Oplossing: Wildcards

```java
public static double totalArea( ArrayList<? extends Shape> arr )
{
    double total = 0;

    for( Shape s : arr )
        if( s != null )
            total += s.area( );

    return total;
}
```

figure 4.31

`totalArea` method revised with wildcards that works if passed an `ArrayList<Square>`

Type erasure

- Generic classes are converted to non-generic classes by compiler: raw classes.
- If generic method is called from which return type has been cleared, a cast is automatically done.

COMPARABLE INTERFACE

`Comparable` interface Java 1.5 (= 5)

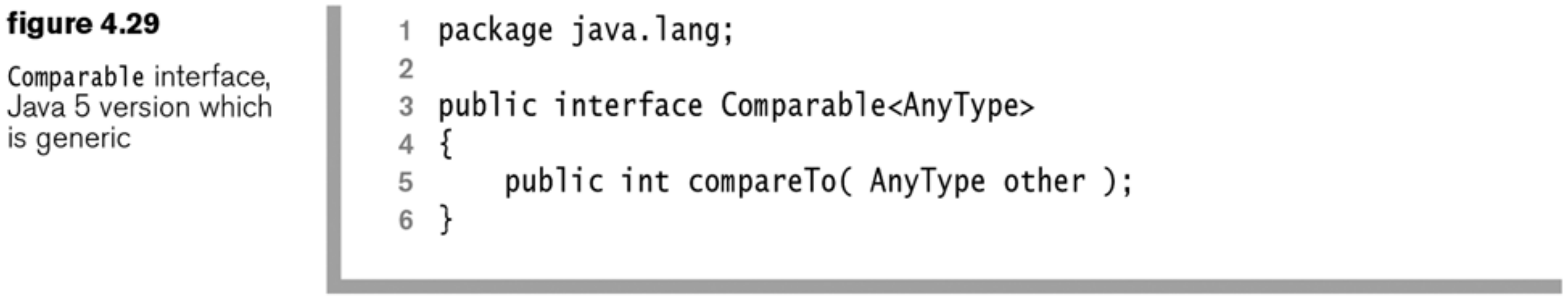


figure 4.29

`Comparable` interface, Java 5 version which is generic

```
package java.lang;

public interface Comparable<AnyType>
{
    public int compareTo( AnyType other );
}
```

- Used to compare objects
- Result `x.compareTo(y):`
  - Als x > y than result > 0
  - Als x < y than result < 0
  - Als x == y than result 0

Voorbeeld van String `compareTo()`

```
String s1 = "Meron";
String s2 = "Merlijn";
System.out.println(s1.compareTo(s2));
```

Output?

Voorbeeld zelfgemaakte compareTo()

```java
public class Shape implements Comparable<Shape> {
    private int area;

    public int getArea(){
        return area;
    }

    @Override
    public int compareTo(Shape other) {
        if (area < other.getArea()) return -1;
        if (area > other.getArea()) return 1;
        return 0;
    }
}
```

Veel gebruikte truc bij compareTo()

```
public int compareTo(Shape other) {
    if (area < other.getArea()) return -1;
    if (area > other.getArea()) return 1;
    return 0;
}
```

Also often written as

```
public int compareTo(Shape other) {
    return area - other.getArea();
}
```

Waar wordt `compareTo()` gebruikt?

- E.g. when sorting:

```
String[] list =
        {"John", "Paul", "George", "Ringo”,
         "Mick", "Keith", "Charlie", "Ron"};
Arrays.sort(list);
```

- Method `sort()` uses `compareTo()` from the `String`-class.