Technische Hochschule Rosenheim
Technical University of Applied Sciences

Object-oriented programming
Chapter 2a – Chained lists

Prof. Dr Kai Höfig

Revision: Fields(1)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Last semester, we learned about fields (arrays) (syntax element `[]`) to store several objects of one type.
- Fields can be used for both primitive data types and non-primitive (reference) data types (i.e. objects of classes, e.g. instances of car). It is important to note that the size must either be specified explicitly upon creation or specified by an initialisation.

```
// create four types of a field with 3 int values therein:
int[] zs1 = {1, 2, 3};
int zs2[] = {1, 2, 3};
int[] zs3 = new int [] {1, 2, 3};
int[] zs4 = new int [3];


System.out.println(zs1.length);  // output in the command line: "3"
System.out.println(zs2.length);  // "3"
System.out.println(zs3.length);  // "3"
System.out.println(zs4.length);  // "3"
```

- In the variable declaration, the [] can either be placed before the identifier (i.e. at the actual data type), or placed after the identifier to define the variable as an array. In the example above, the variables `zs1`, `zs2` and `zs3` were statically initialised with the values 1, 2 and 3, whereas `zs4` was created with `new`, whereby the 3 values were initialised with the default values; these are `false` for truth (Boolean) values, `0` for numerical values and `null` for non-primitive (reference) data types.

Revision: Fields(2)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- The read and write access are now with the [] operator, this time specifically after the identifier. The length of a field can always be found out from the `.length` attribute that is managed by the JVM. The `for-each` loop has a special role, where the JVM takes over the array access, but this can only be read access.

```
int[] zs = new int [3];
zs[1] = 1337;
zs[2] = zs[0] - zs[1];

for (int i = 0; i < zs.length; i++)
    System.out.println(zs[i]);  // "0 1337 -1337"

for (int z : zs) {
    System.out.println(z);  // "0 1337 -1337"
    z = 5;  // no syntax error, but zs remains unchanged!
}
```

Lists as sequential data structures

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Since arrays cannot grow or shrink dynamically, we will now develop our own data structure that can do that, a list.

- To do so, we first declare an interface, in which we declare the required interactions of objects of our list type. We are dealing with a list of int values here.

```
public interface IntList {
  // according to the [] operator:
  int get(int i);
  void put(int i, int v);

  // regarding the list length
  void add(int v);
  void remove(int i);

  int length();
}
```

- In doing so, it is important that the methods of the interface have *no* implementation -- the method declaration ends after the signature with a semicolon.

- The main advantage of interfaces is that they can be used without any knowledge of their *implementation*

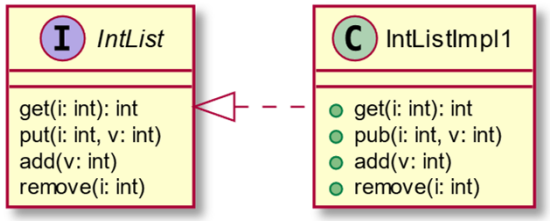
Realisation and *is-a* relationship

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- If we now want to write a class that fulfils this interface, we use the keyword `implements` in the class definition, and implement the prescribed methods.

```
public class IntListImpl1 implements IntList{
  public int get(int i)          { /* TODO */ return 0;}
  public void put(int i, int v) { /* TODO */ }
  public void add(int v)         { /* TODO */ }
  public void remove(int i)      { /* TODO */ }
  public int length()           { /* TODO */ return 0;}
}
```

- This *realisation* is shown in UML with a dashed line arrow and an empty triangle tip:



- Example:

```
IntList li = new IntListImpl1();
System.out.println(li instanceof IntList);  // "true"
```

List realised with array

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- We can now use the existing array data structure in order to create such a list data structure.

```
public class IntListImplArray implements IntList{
  private int[] zs;

  public IntListImplArray() {
    zs = new int [0]; }  // empty.

  public int get(int i) {
    return zs[i];
    }

  public void put(int i, int v) {
    zs[i] = v;
    }

  public void add(int v) {
    int[] newInt = new int [zs.length + 1];
    System.arraycopy(zs, 0, newInt, 0, zs.length);
        newInt[zs.length] = v;
    zs = newInt;
    }

  public int remove(int i) {
    int r = zs[i];
    int[] newInt = new int [zs.length - 1];
    for (int j = 0, k = 0; j < zs.length; j++) {
      if (j == i) continue;
            newInt[k++] = zs[j];
        }
    zs = newInt;
    return r;
    }

  public int length() {
    return zs.length;
    }

  public String toString() {
    return Arrays.toString(zs);
    }
}
```

Improved list with array and block size

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Because the implementation described above constantly creates new arrays every time there is write access, this implementation quickly becomes inefficient. An alternative possibility is to use a block size `BS`.

```
public void add(int v) {
  if (len < zs.length) {
    zs[len++] = v;  // include in count!
    return;
    }
  int[] newInt = new int [zs.length + BS];
  System.arraycopy(zs, 0, newInt, 0, zs.length);
    newInt[len++] = v;  // continue counting!
  zs = newInt;
}
```

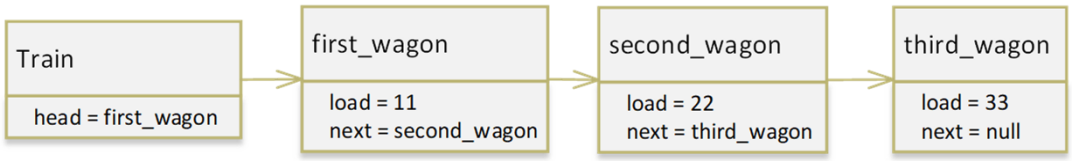
Chained list

Technische Hochschule Rosenheim
Technical University of Applied Sciences

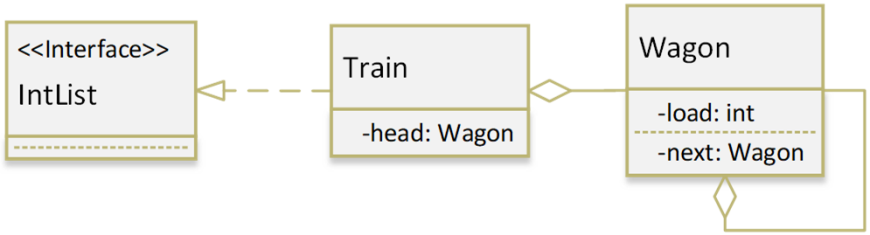
- The implementations above are
  - fast for read and write access, but
  - inefficient for deleting

- We will now look for another implementation, using the example of a train. In abstract terms, a freight train is a list:

  - If the locomotive travels alone, the list is empty.
  - We can easily incorporate new wagons (*insert*), remove existing wagons (*delete*) or add new wagons (*add*).
  - For example, if we want the contents of the 3rd wagon, we start at the front at the locomotive and "work our way down" to the 3rd wagon.
  - If we were to draw a "freight train" with loads of 11, 22 and 33 as an object diagram, it could look like this:

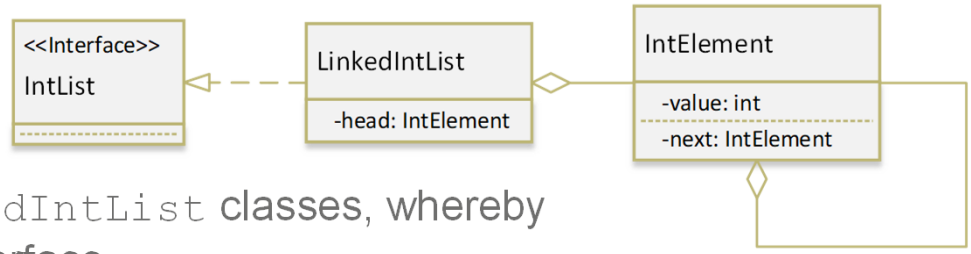


  - ..and as a class diagram, like this

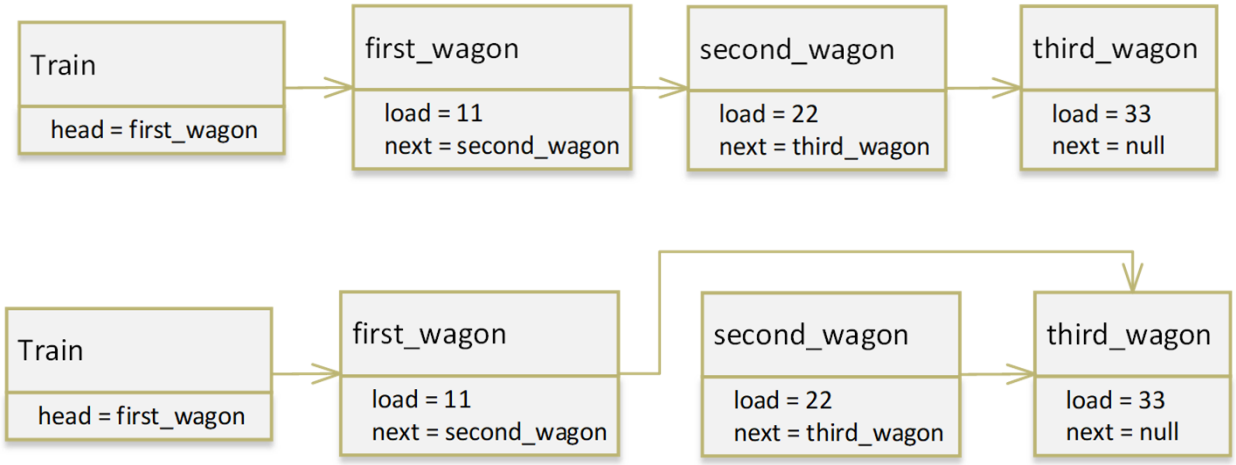


More abstract: a chained list of integers (whole numbers)

Technische Hochschule Rosenheim
Technical University of Applied Sciences



- We need the `IntElement` and `LinkedIntList` classes, whereby the latter implements the `IntList` interface.
- The methods that we need for interaction must enable insertion, deletion, reading and writing.
- Now the deletion of an element is a bit more tricky. In this case, an element is not deleted directly, but instead is made inaccessible by changing the link (which in turn causes the deletion by the JVM).



Here, the second wagon is removed by "skipping" it.

Summary

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Although **fields** (arrays) have fast read and write access, they are not suitable for variable volumes of data or insert and delete operations, due to their unchangeable size
- An **array-based list** with block-by-block allocation can be useful, especially if we want to add, read and write.
- Although a **chained list** has slower read access, it can insert, add and delete very efficiently; this makes it particularly suitable for data processing where the data streams are processed sequentially and the number of elements to be expected is unknown.
- **Working with interfaces** ensures that the underlying implementation can be replaced at any time. In this way, responsibilities can be clearly allocated when developing in a team.