CHAPTER 6: GENERICS

LEARNING OBJECTIVES

- Know the advantages of type-safe data structures
- Be able to instantiate and use parameterisable data types
- Know and be able to use collection classes and associated interfaces
- Understand why collection interfaces must also be generic
- Be able to use and implement generic interfaces
- Understand and be able to apply the iterator concept
- Be able to define and use your own classes and interfaces in a parameterisable way

6.1 USING PARAMETERISED CLASSES

MOTIVATION

`java.util.ArrayList` can be used to realise an array of squares with a variable number of elements.

```
ArrayList squares = new ArrayList();
squares.add(new Square(1.0));
squares.add(new Square(2.0));
```

When the `ArrayList` class is instantiated in this way, the add method expects a parameter of type `Object`.

This means that the following code is correct as well:

```
ArrayList squares = new ArrayList();
squares.add(new Square(1.0));
squares.add(new Student(4711, "Codie Coder")); // add student to list
```

When iterating over the list we will probably run into problems since the list elements are not all squares!

How can we create type-safe objects, e.g. for dynamic data structures like lists?

Let's assume we want to output the area of each square in the list. `get(int)` returns an object reference, so we apply a class cast to convert the reference to `Square`.

```
for (int i = 0; i < squares.size(); ++i)
    System.out.println(((Square) squares.get(i)).area());
```

A `ClassCastException` will be thrown for the second element which is of type `Student`.

The remedy here is a type query with `instanceof`:

```
for (int i = 0; i < squares.size(); ++i)
    if (squares.get(i) instanceof Square)
        System.out.println(((Square) squares.get(i)).area());
```

The `instanceof` query does not solve the actual problem, which is the lack of type safety. The list should "know" that it can only accept square references!

As a solution to this problem, classes and interfaces were made parameterisable. For our example this means that we define our ArrayList object with class `Square` as so called type parameter.

This allows the compiler to check at comple time (not at runtime!) whether an add method is correctly parameterised with a square reference:

```
ArrayList<Square> squares = new ArrayList()<Square>;
squares.add(new Square(1.0)); // ok
squares.add(new Square(2.0)); // ok
squares.add(new Student(4711, "Codie Coder")); // compile error!
```

The output loop is also simplified. If `ArrayList` is instantiated with type `Square`, the signature of the `get` and `add` methods change to:

```
Square get(int index);
void add(Square value);
```

Thus get already returns the correct type, a class cast is not necessary any more:

```
for (int i = 0; i < squares.size(); ++i) {
    System.out.println(list.get(i).area());
}
```

The above code works because ArrayList is defined as a parameterisable class with `E` being the type parameter (see Java API):

```
public class ArrayList<E> {
    public E get(int index) { ... }
    void add(E value) { ... }
    ...
}
```

At compile time the compiler replaces `E` with the type parameter given, e.g. `Square`.

REMARKS

- If the type parameter is omitted from a parameterisable class such as ArrayList, the "raw type" is used and the compiler displays a warning. The warning can be suppressed using the annotation @SuppressWarnings("rawtypes").
- The type parameter can be omitted for declarations with simultaneous initialisation:

```
ArrayList<square> list = new ArrayList()<>;
```

- Classes and interfaces can have several type parameters. For example, the HashMap class has two type parameters:

```
class HashMap<K, V>
```

- Generics can only be used with objects, so autoboxing and autounboxing are required for primitive types in lists:

```
ArrayList<Integer> list = new ArrayList<>();
list.add(2); // Autoboxing int -> Integer
int firstInt = list.get(0); // Autounboxing Integer -> int
```

EXERCISE: USING PARAMETERISED CLASSES

Which statements are correct?

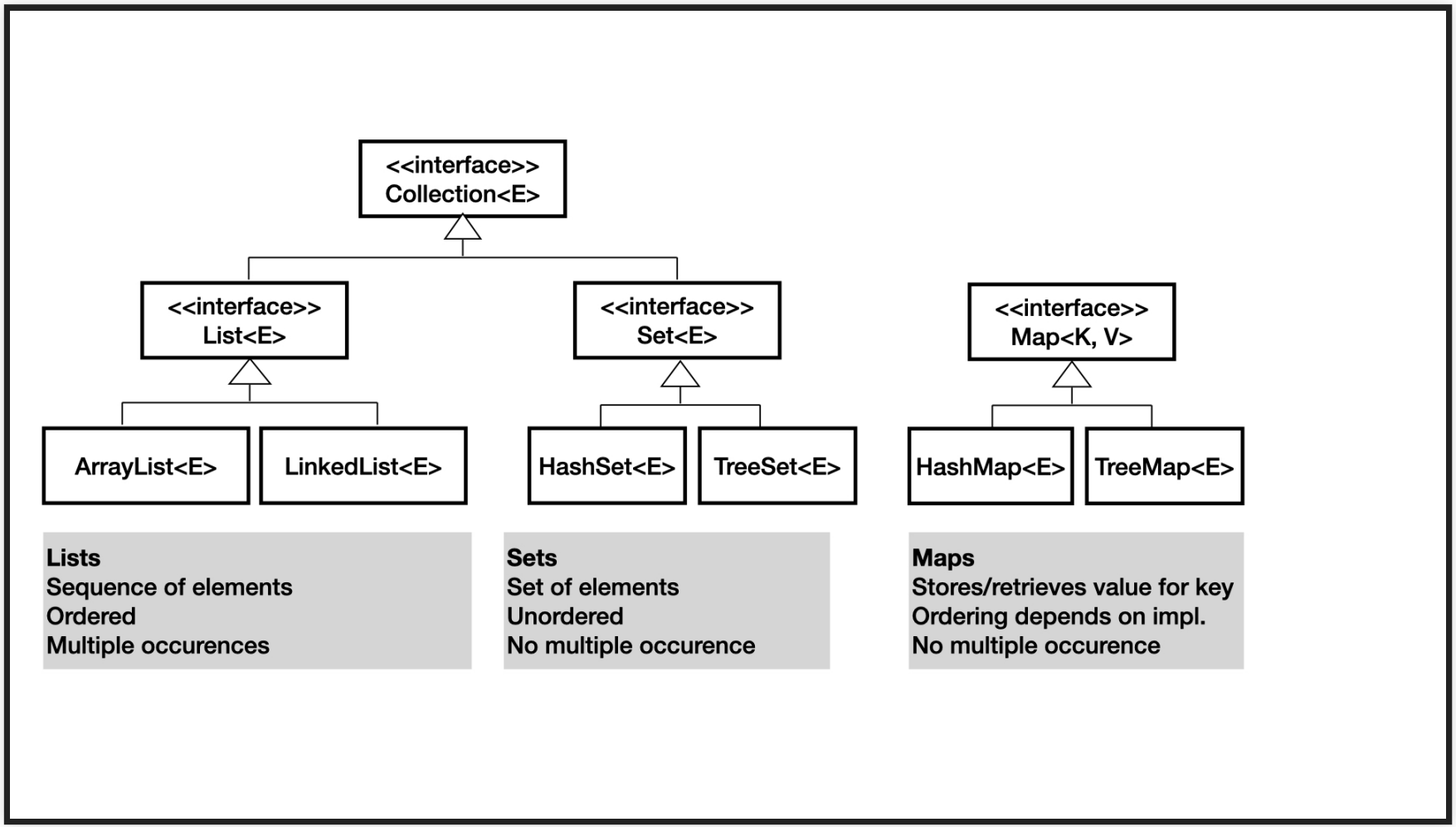
- [ ] If you omit the type parameter, the Java compiler reports an error.
- [ ] The use of type parameters prevents runtime errors because the runtime system ignores unsuitable objects.
- [ ] Type parameters enable type checking by the Java compiler and thus prevent runtime errors.
- [ ] The get method of an ArrayList returns an object reference in the untyped case and a T reference in the typed case (T is the type parameter)

6.2 COLLECTION CLASSES

The Java runtime library offers parameterizable classes in the `java.util` package for efficient storage of data in frequently used data structures such as lists, hash tables or trees. In addition to the classes, there are also a number of parameterizable interfaces that define common methods for the classes and thus make Java programs more flexible.

An introduction to hash tables can be found here, for example: tutorialspoint/hashtables.

The following UML model provides an overview of inheritance and implementation relationships (only selected classes).



COLLECTIONS

A `Collection<E>` is an unordered group of elements of class E (E is the type parameter).

Important operations on an object of type `Collection<E>` are:

- `boolean add(E e)`
  Add e to collection
- `void clear()`
  Remove all elements from collection
- `boolean contains(Object o)`
  Yield true if collection contains reference o
- `Iterator<E> iterator()`
  Returns an iterator object for iteration over collection
- `boolean remove(Object o)`
  Removes reference o, returns true on success
- `int size()`
  Yields number of elements in collection

LIST<E>

List implementations manage class E object references in the form of linear lists. They offer the collection methods, moreover elements can be inserted or deleted anywhere. `ArrayList<E>` offers high-performance read access, while `LinkedList<E>` offers high-performance insertion and deletion of elements. Lists allow elements to appear multiple times and entries are added to the end of the list using `add(E e)` or inserted at a specific position using `add(int index, E e)`.

Operations on Lists in addition to those inherited from Collection:

- `void add(int index, E e)`
  Inserts e into the index-th position
- `E get(int index)`
  Returns the element at position `index`
- `E remove(int index)`
  Deletes element at position `index` and returns the deleted element

SET<E>

Sets store sets of object references of type E, where each reference can only occur once. Implementation of the set interface are:

- `HashSet<E>`: stores elements of type E in random order
- `TreeSet<E>`: stores elements of type E in ascending order

A Set provides the operations (inherited from Collections)

- `boolean add(E e)`
  Adds e if it is not already present
- `void clear()`
  Removes all elements
- `boolean contains(Object o)`
  Returns true if reference o is present
- `Iterator<E> iterator()`
  Returns iterator object for iteration
- `boolean remove(Object o)`
  Deletes o if present and returns true, false otherwise
- `int size()`
  Returns number of elements

Example: Generation and output of lottery numbers in ascending order

```
class Lottery {
    public static void main(String[] args) {
        TreeSet<Integer> drawing = new TreeSet<Integer>();
        // 6 (out of 49) numbers are required
        while (drawing.size() < 6) {
            // only new numbers will be added!
            drawing.add((int) (Math.random() * 49) + 1);
        }
        // Output all 6 numbers
        System.out.println("Lotto numbers: " + drawing);
    }
}
```

Question: What would change if we used a HashSet instead of a TreeSet?

MAP<K, V>

A Map<K, V> is a data structure that stores any number of pairs of keys of type K (key type) and values of type V (value type). K and V can be of any class. It is important that the keys used are unique. Take the example of student registration numbers at a university.

For example, consider an object that implements the map interface:

```
// keys are Integers, compiler derives HashMap parameters from left side
Map<Integer, Student> students = new HashMap<>;
```

Students can be inserted with their registration numbers as keys as follows:

```
students.put(4711, new Student(4711, "Codie Coder");
students.put(4712, new Student(4712, "Bart Simpson");
```

Retrieve Codie's record for registration number 4711:

```
Student codie = students.get(4711);
```

The Map implementations considered here are HashMap and TreeMap. HashMap stores elements in unordered order, TreeMap stores them in ascending order. Examples of creating map objects:

```
Map<Integer, Student> students = new HashMap<>();
Map<Integer, Student> students = new TreeMap<>();
```

Which implementation is chosen depends on the requirements in terms of element ordering or performance (HashMap is faster than TreeMap).

Important operations of a Map<K, V> are:

- `void clear()`
  Clears all key-value pairs
- `V get(K key)`
  Returns value for key or null if key does not exist
- `V put(K k, V value)`
  Inserts value for key k
- `int size()`
  Returns the number of key-value pairs
- `V remove(Object key)`
  Deletes the key-value pair for key
- `Set<K> keySet()`
  Returns the keys as a set, e.g. for iteration
- `Collection<V> values()`
  Returns all values as a collection, e.g. for iteration

EXAMPLE: STORING CONTACT INFORMATION IN A MAP

Given the following class

```
class Contact {
    private String name, city, street;
    private int postCode, nr;

    public Contact(String name, ...) { ... }

    public String getName() { return name; }
    public String toString() { return "Contact [" + name + ... }
}
```

Multiple `Contact` instances could be stored in a map like this:

```
Map<String, Contact> contacts = new TreeMap<>();
Contact peter = new Contact("Peter", 85049,
    "Ingolstadt", "Esplanade", 10);
Contact sarah = new Contact("Sarah", 54321,
    "Javatown", "Polymorphic road", 1);
// store contacts using name as key
contacts.put(sarah.getName(), sarah);
contacts.put(peter.getName(), peter);
```

A list of contacts can be produced as follows:

```
for (Contact contact : contacts.values())
    System.out.println(contact); // call contact.toString() implicitely
```

Sarah's contact information can be retrieved as follows:

```
Contact sarah = contacts.get("Sarah");
if (sarah != null)
    System.out.println("Sarah's contact information: " + sarah);
else
    System.out.println("Sarah not found!");
```

EXERCISE: COLLECTION CLASSES

Which statements are correct?

- ☐ Lists and sets are ordered and indexable.
- ☐ List, Set and Map specify methods that must be implemented by the concrete classes (e.g. HashMap).
- ☐ Sets are used if no identical elements are to be stored.
- ☐ Lists are ordered and indexable, sets and maps are generally not.

Which statements are correct?

- [ ] It makes no difference whether you use TreeSet or HashSet.
- [ ] The advantage of maps over sets or lists lies in the high-performance read access via a key.
- [ ] HashSet or HashMap cause an ascending order of the stored elements or keys.

EXERCISE: ADVANTAGES AND DISADVANTAGES OF DATA STRUCTURES

Given are test classes for `ArrayList`, `LinkedList`, `TreeSet`, `HashSet`, `TreeMap` and `HashMap`. Add the methods for working with the data structures and test the advantages and disadvantages using the example.

Unzip collection-sim.zip and import the folder as "Existing Maven Project".

Answer the following questions:

- What are typical methods for working with the data structure?
- What are the advantages of the corresponding data structure? (In particular, consider the time measurements)
- How is the data structured internally? (Take a look at the Java source code or search the Internet)

6.3 GENERIC INTERFACES

We already know four parameterisable interfaces: `Collection<E>`, `List<E>`, `Set<E>` and `Map<K, V>`.

In the following chapter, we will take a look at the interfaces `Comparable<E>` and `Comparator<E>`, as well as introduce the interfaces `Iterable<E>` and `Iterator<E>`.

EXAMPLE: COMPARING STUDENT OBJECTS

```
class Student implements Comparable<Student> {
  private int registerNumber;
  private String name;
  // ... CTOR etc.
  public int compareTo(Student s) { // comparison by registerNumber
    return registerNumber - s.registerNumber;
  }
}
```

In contrast to the non-parameterised version, the parameter is of type Student, i.e. neither type checking nor typecast is required. The compiler can guarantee whether a suitable reference is passed (student or subclasses thereof) eliminating runtime conversions and checks.

EXAMPLE: COMPARATOR FOR SQUARES

```
class SideComparator implements Comparator<Square> {
  public int compare(Square sq1, Square sq2) {
    double perimeter1 = sq1.perimeter();
    double perimeter2 = sq2.perimeter();

    if (perimeter1 < perimeter2)
        return -1;
    else if (perimeter1 > perimeter2)
        return 1;
    else
        return 0;
  }
}
```

The same approach can also be used for `Comparator`, i.e. no typecast or type check is necessary at runtime.

ITERATIONS OVER CUSTOM DATA TYPES

To iterate over collections based on your own classes, classic while, do-while and for loops could be used using element counts and index-based accesses.

What if we just need a convenient loop over all elements of such a collection?

The parameterizable interfaces `Iterable<T>` and `Iterator<T>` are available for this in Java.

- `Iterable<T>`
  an iterable object provides an iterator method that creates an iterator object

```
interface Iterable<T> {
    Iterator<T> iterator();
}
```

- `Iterator<T>`
  an iterator object allows iterating over all elements of the collection using the hasNext and next methods

```
interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

EXAMPLE ITERATION

In the following example an ArrayList is created and three elements are added. For iterating the ArrayList

- a new iterator is created using `stringListe.iterator();`
- then we iterate through the list using `hasNext()` and `next()`:

```
List<String> stringList = new ArrayList<>();
stringList.add("Java");
stringList.add("is");
stringList.add("super");
Iterator<String> iterator = stringList.iterator();
while (iterator.hasNext())
  System.out.println(iterator.next());
```

By parameterizing the interface, next already supplies the correct type (string), so typecasts are eliminated and the compiler can check the correct usage at compilation time.

This can be done more quickly with the so-called for-each loop in Java, which automatically uses the iterator.

```
for (String text : stringList)
  System.out.println(text);
```

EXAMPLE OF YOUR OWN ITERATOR

Step 1: An implementation of the iterator that can work through a data structure.

```
class IntListIterator implements Iterator<Integer> {
    private int index = 0; // for iteration, start at 0
    private List<Integer> list; // List to iterate over

    public IntListIterator(List<Integer> list) {
        this.list = list;
    }
    public boolean hasNext() {
        return index < list.size();
    }
    public Integer next() {
        return (hasNext() ? list.get(index++) : null);
    }
}
```

Step 2: An implementation of the iterable that creates the iterator.

```
class IntList implements Iterable<Integer> {
    private List<Integer> list = new ArrayList<>();

    public IntList add(int value) {
        list.add(value);
        return this; // for chaining add calls, see application
    }

    public Iterator<Integer> iterator() {
        return new IntListIterator(list); // Create iterator!
    }
}
```

Step 3: Application

```
// Application: since add() returns the list, we can chain calls
IntList l = new IntList().add(14).add(1).add(100);

for (int i : l)
    System.out.println(i);
```

Explanation of the last example:

- IntList creates a new iterator for the iteration.
- This returns all elements of the list, starting at index 0.
- The for-each loop is translated by the compiler into the well-known construct:

```
Iterator<Integer> iterator = l.iterator();
while (iterator.hasNext()) {
    Integer i = iterator.next();
    System.out.println(i);
}
```

EXERCISE: GENERIC INTERFACES

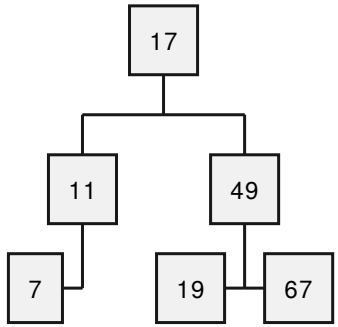
Which statements are correct?

- [ ] Iterable objects provide the Iterable interface.
- [ ] With Comparator it makes no difference whether you use the type parameter or not.
- [ ] Every iterator class should implement the hasNext() and next() methods.
- [ ] Using Comparable with type parameter saves a typecast in the implementation of the compareTo method.
- [ ] No iterator class is necessary for your own iterable classes.

6.4 HOW TO PARAMETERIZE YOUR OWN CLASSES

Consider the following implementation of a binary search tree.

- Each tree element consists of a data part of type int and a left and right subtree each of type `BinTree` (recursive data structure).
- The `add` method sorts `value` into either the right or left subtree, depending on whether `value` is smaller or larger than the root element `data` of the currently called tree.
- `add` should return the tree itself in order to allow chaining
- The binary tree shown illustrates this principle. Please note that `add` is called at the root element of the tree.



BINARY TREE WITHOUT PAREMETERIZATION

```
public class BinTree {
    private int data;               // payload for this node
    private BinTree left, right;    // left and right subtree

    public BinTree(int data) {
        this.data = data; this.left = null; this.right = null; // store data, no subtrees
    }
    public BinTree add(int value) {
        if (value < data) {                 // value must be stored in left subtree
            if (left == null)
                left = new BinTree(value);  // new node in left subtree
            else
                left.add(value);            // add recursively to left subtree
        } else if (value > data) {          // value must be stored in right subtree
            if (right == null)
                right = new BinTree(value); // new node in right subtree
            else
                right.add(value);           // add recursively to right subtree
        }
        return this;
    }
    public static void main(String[] args) {
        BinTree tree = new BinTree(17);
        tree.add(49); // will add in right subtree since 49 > 17
    }
}
```

The BinTree class can store any number of `int` values. However, if `string` values are to be saved instead of `int` values, the implementation must be adapted in "copy&paste" style. This procedure is error-prone and not easy to maintain. The solution to this problem is to parameterise the class.

BINARY TREE WITH PAREMETERIZATION

```
public class BinTree<T extends Comparable<T>> {
    private T data;                     // payload of this node
    private BinTree<T> left, right;     // left and right subtree

    public BinTree(T data) {
        this.data = data; this.left = null; this.right = null; // store data, no subtrees
    }
    public BinTree add(T value) {
        if (value.compareTo(data) < 0) {        // value must be stored in left subtree
            if (left == null)
                left = new BinTree<T>(value);   // new node in left subtree
            else
                left.add(value);                // add recursively to left subtree
        } else if (value.compareTo(data) > 0) { // value must be stored in right subtree
            if (right == null)
                right = new BinTree<T>(value);  // new node in right subtree
            else
                right.add(value);               // add recursively to right subtree
            }
        }
        return this;
    }
    public static void main(String[] args) {
        BinTree<Integer> tree1 = new BinTree<>(17).add(49); // will add to right subtree since 49 > 17
        BinTree<String> tree2 = new BinTree<>("Java").add("is cool!");
    }
}
```

Explanation:

- The BinTree class is parameterised with the type `T`. There is an additional condition for `T`: `T extends Comparable<T>`. This means that only parameter classes that implement the Comparable interface are allowed. The condition is necessary because the add method needs a comparison operator on the data of type `T` to insert the new value.
- In many cases it is sufficient to parameterise without a condition as described above. This makes it relatively easy to parameterise a given class: put the parameter (an uppercase letter, e.g. `T`) after the class name and use `T` wherever necessary in the class definition. The same is true for interfaces.

EXERCISE: PARAMETERISING YOUR OWN CLASSES

Which statements are correct?

- [ ] Parameterised classes help to avoid "copy&paste" and make programs easier to maintain.
- [ ] A binary tree is so called because each tree node has exactly two children.
- [ ] T extends Comparable<T> means that class T inherits from Comparable.
- [ ] T extends Comparable<T> means that the type T implements the interface Comparable.

6.5 EXERCISE

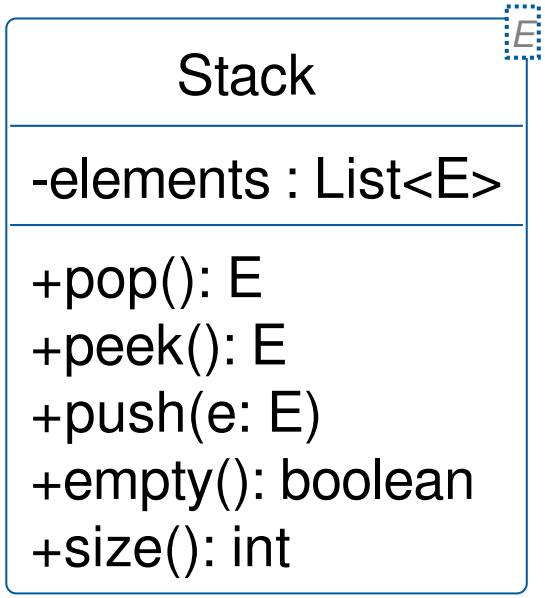
PARAMETERIZED CLASS STACK

In the following exercise, a last-in-first-out stack for any data type is to be created using a parameterised class.

Unzip stack.zip and import the folder as "Existing Maven Project".

EXERCISE: STACK

Extend the existing Stack class according to the following UML class model.



- Parameterise the class `Stack` and add the type `E`
- Add a private attribute `elements` of type `java.util.List` that uses the type `E`.
- Add a constructor that initialises the attribute `elements` with a new instance of the class `java.util.ArrayList`, use the generic type `E`.
- Add a method `size()` which returns the number of elements with the help of the `List` attribute
- Add a method `empty()` which returns `True` if the number of elements is 0

- Add a method `push()` which adds an entry to the attribute `elements` using the method `add(E e)`.
- Add a method `peek()`, which should return `null` if the stack is empty, otherwise use the method `get(int index)` to return the last entry of the list
- Add a method `pop()`, which should return `null` if the stack is empty, otherwise use the method `remove(int index)` to remove and return the last entry of the list

ÜBUNG: STACK-INITIALISIERUNG

Which initialisations are correct and work?

- [ ] `Stack myStack = new Stack();`
- [ ] `Stack<Stack> myStack = new Stack();`
- [ ] `Stack<int> myStack = new Stack<int>();`
- [ ] `Stack<String> myStack = new Stack<String>();`
- [ ] `Stack myStack = new Stack<Stack>();`
- [ ] `Stack<boolean> myStack = new Stack<>();`
- [ ] `Stack<Integer> myStack = new Stack<>();`

EXERCISE: USING CLASS STACK

Given the following code, which statements are true?

```
Stack<Integer> numbers = new Stack<>();
numbers.push(3);
numbers.push(1);
```

- ☐ `numbers.pop()` yields `3`
- ☐ `numbers.push(4); numbers.peek(); numbers.pop();` yields `4`
- ☐ `numbers.peek()` yields `1`
- ☐ `numbers.peek()` yields `3`
- ☐ `numbers.push(4); numbers.pop(); numbers.peek();` yields `1`
- ☐ `numbers.pop()` yields `1`