7 Generic Programming

- Generics
- Collection Framework

Generics: methods and classes can have *types as parameters*.

- Another form of defining polymorphic methods
- Alternative to **`Interface`** for implementing comparable accesses with different classes.
- Goal: generic formulation of algorithms that will be later instantiated depending on the type.
- Syntax with methods: type parameters are given before the return type of the method

```
public <T> T doSomething ( T[] array, int i )  {
   return array [i];
}
...
Integer [] IntegerArray;
...
doSomething (IntegerArray, 5);
...
Double []  DoubleArray;
...
doSomething (DoubleArray, 4);
```

Example with generic method:

```
public class K7B01E_GenericMethod {

  public static < T > String convert( T[] array ) {
    String result="|";
    for ( int i=0; i< array.length; i++)
        result += "␣" + array[i] + "␣|";
    return result;
  }

  public static void main( String args[] ) {
    Integer[]   iArray = {3, 5, 7 };
    Double[]    dArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] cArray = { ’H’, ’E’, ’L’, ’L’, ’O’ };

    System.out.println(
       convert( iArray ) + "\n\n" +
       convert( dArray ) + "\n\n" +
       convert( cArray ) );
  }
}
```

Why Generics?

```
Object o = "String";
String s = (String) o;
```

- Explicit type cast: object **o** must include String object.
- This is ok here...

But:

```
Object o = Integer.valueOf( 42 );    // or Autoboxing: o = 42;
String s = (String) o;
```

- Type cast implies check for class compatibility at run time, thus here: run time error...
- Alternatives: catch exception or use **instanceof**
- Better: check at compile time, then no need for test at run time, run time errors impossible.

$\leadsto$ **Generics**

Example: generic method for maximum,
`java.lang` includes **`interface Comparable<T>`**

```
public class K7B02E_GenericMaximum  {

 public static <T extends Comparable<T> >
                     T maximum( T x, T y, T z ){
   T max = x;
   if ( y.compareTo( max ) > 0 )
      max = y;
   if ( z.compareTo( max ) > 0 )
      max = z;
   return max;
 }

 public static void main( String args[] ) {
   System.out.println(
       maximum(8, 6, 4) + "␣␣"
     + maximum(1.1, 7.7, 4.4) + "␣␣"
     + maximum( "Pear", "Apple", "Orange" ) );
 }
}
```

- At compilation time, the type variables are removed ('erasure') and replaced by concrete types.
- what remains are 'objects' (i.e., **`Object`** as base type); for a generic type **`<T1 extends T2>`** the base type **`T2`** is chosen..
- The purpose of generics is especially checking of type safety during compilation time.

```java
public static <T extends Comparable<T> >
                  T maximum( T x, T y, T z ){
   T max = x;
   if ( y.compareTo( max ) > 0 ) max = y;
   if ( z.compareTo( max ) > 0 ) max = z;
   return max;
}
```

is compiled to:

```java
public static
 Comparable maximum (Comparable x, Comparable y, Comparable z ){
   Comparable max = x;
   if ( y.compareTo( max ) > 0 ) max = y;
   if ( z.compareTo( max ) > 0 ) max = z;
   return max;
}
```

Syntax of the header of **Generic Classes**:

```
class name<T1, T2, ..., Tn> ... {...}
```

Style convention recommends parameter names for generic parameters:

- **E** - Element (esp. with Java Collections Framework, see later)
- **K** - Key
- **N** - Number
- **T** - Type (general type)
- **V** - Value
- **S,U,V** etc. – second, third, fourth type...

Example: generic stack (as static class in `K7B03E_GenericStack`)

```java
  public static class GenericStack< E > {
    private int top;
    private E[] elements;

    public GenericStack() {
       top=-1;
       elements = ( E[] ) new Object[ 10 ];
     //elements = new E[ 10 ];
    }

    public boolean isFull(){return (top == 9);}
    public boolean isEmpty(){return (top == -1);}
    public void push(E value){elements[++top] = value;}
    public E pop(){return elements[top--];}
  }
```

compiler warning at:

```
elements = ( E[] ) new Object[ 10 ];
```

System cannot guarantee type safety, here only as warning!

```
Note: K7B03E_GenericStack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

(obvious) alternative does not work:

```
elements = new E[ 10 ];
```

with error message

```
K7B03E_GenericStack.java:10: error: generic array creation
     elements = new E[ 10 ];
```

The warning can be ignored here since the array is accessed only through **push** and **pop**, but no external method modifies the array and could insert objects of wrong types.

```
public class K7B03E_GenericStack {
 public static class GenericStack< E > {...} \\s.o.

 public static void main( String args[] ) {
   double[] dElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
   int[] iElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   GenericStack < Double > dStack
                       = new GenericStack< Double >();

   GenericStack < Integer > iStack
                       = new GenericStack< Integer >();

   for ( int i=0; i< dElements.length; i++)
     if (!dStack.isFull()) dStack.push( dElements[i]  );
   while (!dStack.isEmpty()) System.out.println( dStack.pop() );

   for ( int i=0; i< iElements.length; i++)
     if (!iStack.isFull()) iStack.push( iElements[i] );
   while (!iStack.isEmpty()) System.out.println( iStack.pop() );
  }
}
```

Since version 7, Java provides *type inference*: Java tries to determine the types from the context.

Then the generic type ‘**`<>`**’ (‘Diamond’) is sufficient, e.g.:

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

al.add(new Double(1.1));
al.add(new Double(2.2));
al.add(new Double(3.3));
for (int i = 0; i < al.size(); i++) {
   System.out.println(a[i]);
}
```

7 Generic Programming

- Generics
- Collection Framework

A ‘collection’ is an aggregation of similar data elements with operations for navigating and/or direct access to the individual elements. The Java language itself only provides arrays for this purpose.

The Collection Framework, part of the package **`java.util`**, supports additional collections in the form of a collection of

- interfaces for typical data structures (e.g. **`Queue`**, **`List`**, **`Set`**), for sequential access (**`Enumeration`**, **`Iterator`**, **`ListIterator`**) and the comparison of elements (**`Comparator`**).
- implementations of the interfaces with different underlying data structures (e.g. **`ArrayList`**, **`Vector`**, **`Stack`** on dynamically extended arrayys, **`LinkedList`** on a doubly linked list).
- algorithms for typical ‘higher’ operations, e.g. **`sort`**, **`binarySearch`**, **`shuffle`**, **`max`**, **`min`** in the class **`Collections`** (for **`ArrayList`**, **`Vector`**, **`Stack`**, **`LinkedList`**, ...) ir **`sort`**, **`binarySearch`** in the class **`Arrays`** (for Arrays).

- **`ArrayList`** and **`LinkedList`** are dynamic sequential data structures.
- Both provide operations expected for **`Array`**, **`Queue`**, **`Stack`** and **`List`**.
- Both can grow and shrink on demand.
- **`ArrayList`** is implemented with dynamic arrays. The operations are very efficient (exception: insertion and deletion inner elements).
- **`LinkedList`** is implemented with a doubly linked list. The operations are usually less efficient than with ArrayLists, but inner insertions and deletions are more efficient.
- **`Vector`** corresponds to **`ArrayList`**, but is synchronized (suitable for accessing from concurrent threads).
- **`Stack`** is a **`Vector`** with explicit **`push`** and **`pop`** operations.
- There are many more classes: **`PriorityQueue`**, **`HashSet`**, **`TreeSet`**, **`EnumMap`**, **`HashMap`**, ...

Since Java 5.0, the Collection Framework supports generics.

- Lists
  - **`ArrayList`** List functionality by mapping to an array
  - **`LinkedList`** doubly linked list
- Sets
  - **`HashSet`** implements the interface **`Set`** with a fast hashing method.
  - **`TreeSet`** implements **`Set`** with a tree that keeps the elements sorted.
  - **`LinkedHashSet`** A fast set implementation that in addition also stores the insertion order of the elements.

- Associative Memory
  - `HashMap` implements an associative memory with a hash method.
  - `TreeMap` Instances of this class keep their elements sorted in a binary tree; implements `SortedMap`.
  - `LinkedHashMap` A fast associative memory that additionally stores the insertion order of its elements.
- Queue
  - `LinkedList` The linked list also implements `Queue`.
  - `ArrayBlockingQueue` A blocking queue.
  - `PriorityQueue` A priority queue.

Short overview of important **`Collection`** methods:

```
interface java.util.Collection<E> extends Iterable
```

- **`Enumeration`**: only **`hasMoreElement()`** and **`nextElement()`**, i.e., enumerated data structure is not changed.
- **`Iterator`**: **`hasNext()`**, **`next()`** and additionally **`remove()`**

For a Collection **`c`** with base type **`E`** the following loops are possible (among others):

```
Iterator<E> it = c.iterator();
while ( it.hasNext() ){ E e = it.next(); ...}
```

(here, **`c`** is not changed) or (for-each-loop, see later):

```
for( E e : c ){ something with e...}
```

Iteration with deletion:

```
Iterator<E> it = c.iterator();
while ( it.hasNext() ){ E e = it.next(); it.remove(); ...}
```

(in this example, **`c`** is empty at the end)

```
boolean add( E obj )
```

Optional. Adds an element to the container and returns **`true`** if it was successfully inserted. Returns **`false`** if an object with the same value is already included and duplicate values are forbidden.

```
void clear()
```

Optional. Deletes all elements in the container.

```
boolean contains( Object obj )
```

Returns **`true`** if the container includes an element equal to **`obj`**.

```
boolean isEmpty()
```

Returns **`true`** if the container does not include any elements.

```
int size()
```

Returns the number of elements in the container.

```
Iterator<E> iterator()
```

Returns **Iterator** object for iterating over all elements of the container.

```
boolean remove( Object obj )
```

Optional. Removes **obj** from the container if it is included.

```
Object[] toArray()
```

Returns array with all elements of the container.

```
<T> T[] toArray( T[] arr )
```

Returns typed array with all elements of the container. Uses **arr** if it is large enough; otherwise, a new array of sufficient size is created.

```
boolean equals( Object obj )
```

Checks if **obj** is also a container and includes the same elements

```
boolean addAll( Collection<? extends E> coll )
```

Adds all elements from the collection **coll** to the container.

```
boolean containsAll( Collection<?> coll )
```

Returns **true** if the container contains all elements of the collection **coll**.

```
boolean removeAll( Collection<?> coll )
```

Optional. Removes all objects from the collection **coll** from the container.

```
boolean retainAll( Collection<?> coll )
```

Optional. Removes all objects that are not contained in collection **coll**.

```
int hashCode()
```

Returns the hash value of the container.

Example: **ArrayList <T>** for **String**

```
import java.util.*;
public class K7B04E_ArrayListString  {
  public static void main(String[] args) {

    ArrayList <String> array = new ArrayList <> ();

    String numbers = System.console().readLine();
    StringTokenizer tokens = new StringTokenizer (numbers);

    while (tokens.hasMoreTokens () )
      array.add ( tokens.nextToken () );

    System.out.println("unsorted: " + array);

    Collections.sort(array);

    System.out.println("sorted:   " + array);

  }
}
```

The compiler checks that **array.add(...)** inserts only objects of type **String**!

Example: `ArrayList <T>` for `Integer`

```java
import java.util.*;
public class K7B05E_ArrayListInteger {
  public static void main(String[] args) {

    ArrayList <Integer> array = new ArrayList <> ();

    String numbers = System.console().readLine();
    StringTokenizer tokens = new StringTokenizer (numbers);

    while (tokens.hasMoreTokens () )
      array.add ( Integer.valueOf(tokens.nextToken () ) );

    System.out.println("unsorted: " + array);

    Collections.sort(array);

    System.out.println("sorted:   " + array);

  }
}
```

The compiler checks that **`array.add(...)`** inserts only objects of type **`Integer`**.