

# Iterator & for cycle

- for (Object o : foo)
  - can be used if foo is an array or **foo can be iterated**
  - how to achieve it?
    - implement the **java.lang.Iterable** interface
- **Iterable** has a single method  
**java.util.Iterator iterator()**
- methods of the **Iterator** interface
  - boolean hasNext()
  - Object next()
  - void remove()
- the iterator typically implemented as an anonymous inner class
- in reality, the Iterator is generic, i.e. **Iterator<T>**
  - we will ignore it for now

# iterator example

```
public class MyArrayWithIterator implements Iterable {  
    private Object[] arr = new Object [5];  
    private int s = 0;
```

```
    public int size() {  
        return s;  
    }
```

```
    public void add(Object o) {  
        ...  
        arr[s++] = o;  
        ...  
    }
```

```
    public Iterator iterator() {  
        return new Iterator() {  
            private int index = 0;  
            public boolean hasNext() {  
                return index < s;  
            }  
            public Object next() {  
                return arr[index++];  
            }  
            public void remove() {  
                throw new  
                UnsupportedOperationException();  
            }  
        };  
    }
```

- since Java 8, remove() is **default**
- the implementation throws this exception

# Assignment 1

- create the interface MyCollection with methods
  - void add(Object o)
  - Object get(int i)
  - void remove(Object o)
  - void remove(int i)
  - int size()
- create an implementation of MyCollection
  - use an array, which is reallocated if needed
  - handle all error states by exceptions
    - access out of bounds of the array
- add the iterator (see previous slides)

# Assignment 2

- create a class representing a balanced binary search tree (e.g., AVL, RB, or any other)
  - for the `int` type
- add the iterator that iterates the tree from the smallest element till biggest one
- create a program, which uses the tree and loads data from arguments of the command-line
  - use `Integer.parseInt(String s)` to transform `String` into `int`
    - do not forget to handle exceptions the method throws in a case, the string cannot be transformed
- think how to update the tree in order it can be defined with the `Object` type
  - i.e. how to achieve that tree elements are comparable
  - implement it

Tests...

# Test 1

- What is printed out – true or false

```
public class Test01 {  
    public static void main(String[] argv) {  
        System.out.println(test());  
    }  
  
    public static boolean test() {  
        try {  
            return true;  
        } finally {  
            return false;  
        }  
    }  
}
```

# Test 2

- What is printed out?

```
public class Test02 {  
  
    public static void main(String[] argv) {  
        try {  
            System.out.println("Hello world!");  
            System.exit(0);  
        } finally {  
            System.out.println("Goodbye");  
        }  
    }  
}
```

# Test 3

- What is printed out

```
public class ParamsTest {
    public ParamsTest(Object o) {
        System.out.println("ParamsTest(Object o)");
    }
    public ParamsTest(long[] a) {
        System.out.println("ParamsTest(long[] a)");
    }
    public static void main(String[] argv) {
        new ParamsTest(null);
    }
}
```

- A cannot be compiled
- B ParamsTest(Object o)
- C ParamsTest(long[] a)



# Test 3

- C is correct answer
- Why?
  - Searching a method/constructor
    - based on the **actual** parameters, all the methods/constructors that can be used, are selected
    - from the selected methods/constructors, the most specific one is selected based on the **formal** parameters
  - **ParamsTest(long[] a)** is more specific than **ParamTest(Object o)**
    - everything, that can assigned to **long[] a** can be also assigned to **Object**
    - but it is not true vice-versa

# Test 4



*Exam test*

- What is printed out

```
class A {
    public static void foo() {
        System.out.println("foo");
    }
}
class B extends A {
    public static void foo() {
        System.out.println("bar");
    }
}
```

```
public class OverloadTest {
    public static void
        main(String[] argv) {
        A a = new A();
        A b = new B();
        a.foo();
        b.foo();
    }
}
```

- A foo bar
- B foo foo
- C bar bar
- D something else

# Test 4

- B is correct
- static methods are not virtual



Slides version PJ03.en.2018.01

This slides are licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).