

Voortgezet Programmeren

Lecture 4: Interfaces and polymorphism

Tommi Tervonen

Econometric Institute, Erasmus School of Economics

Rule #1: never duplicate code

```
public class Course {  
    public Course(String name) { ... }  
    public void addStudent(Student s) { ... }  
    public Student[] getStudents() { ... }  
}
```

```
public class Student {  
    public Student(String name) { ... }  
    public String getName() { ... }  
}
```

```
public class Sorter {  
  
    public static void insSort(Student[] arr) {  
        for (int j=1;j<arr.length;j++) {  
            int key = arr[j].getName().length();  
            int i = j-1;  
            while (i >= 0 &&  
arr[i].getName().length() > key) {  
                arr[i+1] = arr[i];  
                i--;  
            }  
            arr[i+1] = arr[j];  
        }  
    }  
}
```

```
public static void insSort(Course[] arr) {  
    for (int j=1;j<arr.length;j++) {  
        int key = arr[j].getStudents().length;  
        int i = j-1;  
        while (i >= 0 &&  
            arr[i].getStudents().length > key) {  
            arr[i+1] = arr[i];  
            i--;  
        }  
        arr[i+1] = arr[j];  
    }  
}
```

Problem: the types are not related

- What we need for sorting is the elements' sorting **key**
- Need a technique for uniformly obtaining it for both Student and Course

// Note: not valid syntax

```
public static void sort(Course[]|Student[] arr) {  
    for (int j=1;j<arr.length;j++) {  
        int key = arr[j].getKey();  
        int i = j-1;  
        while (i >= 0 &&  
            arr[i].getKey() > key) {  
            arr[i+1] = arr[i];  
            i--;  
        }  
        arr[i+1] = arr[j];  
    }  
}
```

- Interfaces define methods that have to be included by classes implementing the interface
- Interfaces declare types like normal classes do: implementing class obtains the interface type

```
public interface Keyable {  
    public int getKey();  
}
```

```
public class Student implements Keyable {  
    private String name;  
  
    public Student(name) { ... }  
    public String getName() { ... }  
  
    public int getKey() {  
        return name.length();  
    }  
}
```

```
public class Course implements Keyable {  
    private Student[] students;  
  
    public Course(name) { ... }  
    public void addStudent(Student s) { ... }  
    public Student[] getStudents() { ... }  
  
    public int getKey() {  
        return students.length;  
    }  
}
```

- Polymorphism means that objects can belong to multiple types and respond to method calls of that type (i.e. be polymorphic)
- Student and Course can act as Keyable

```
Keyable k1 = new Student("tommi");  
Keyable k2 = new Course("FEB23007");  
int key1 = k1.getKey();  
int key2 = k2.getKey();  
Student s2 = new Student("tommi2");  
int key3 = s2.getKey();
```

```
public static void sort(Keyable[] arr) {  
    for (int j=1;j<arr.length;j++) {  
        int key = arr[j].getKey();  
        int i = j-1;  
        while (i >= 0 &&  
arr[i].getKey() > key) {  
            arr[i+1] = arr[i];  
            i--;  
        }  
        arr[i+1] = arr[j];  
    }  
}
```

```
Keyable[] arr = new Keyable[] {  
    new Student("tommi"),  
    new Course("c1"),  
    new Course("c2")  
};
```

```
Sorter.insSort(arr);
```

- All method implementations have to be bound to their implementations (i.e. the actual code that gets executed)
- With polymorphic objects (in Java with all objects) the actual implementation is determined only run-time through *dynamic binding*
- Compare with `static`

- Similarly to compatible primitives (i.e. int/double), also object variables can be cast to other types
- **Upcasting** occurs automatically when casting to implemented interface (similarly to `double x = 0;`)
- **Downcasting** has to be made explicitly

```
Keyable k1 = new Student("tommi");  
int key1 = k1.getKey();  
Student s2 = new Student("tommi2");  
String name2 = s2.getName(); // ok  
String name1 = k1.getName();  
// ^ syntax error: k1 != a Student  
Student nk1 = (Student) k1;  
n1 = nk1.getName(); // ok
```

ClassCastException and instanceof

- Beware: you should know what you're downcasting

```
Keyable k1 = new Student("tommi");  
Course c = (Course) k1; // compiles fine  
// but throws ClassCastException on run-time
```

- instanceof allows to find whether an object is an instance of a class

```
k1 instanceof Student; // true  
k1 instanceof Keyable; // true  
k1 instanceof Course; // false
```

```
public String getKeyableName(Keyable k) {  
    if (k instanceof Student) {  
        Student s = (Student) k;  
        return s.getName();  
    } else if (k instanceof Course) {  
        Course c = (Course) k;  
        return c.getName();  
    } else { // some other Keyable  
        return Integer.toString(k.getKey());  
    }  
}
```

Example interfaces: Iterable<T> / Iterator<T>

```
package java.lang;

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

public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

- Belong to the Java Collections Framework
- Often implemented in an *inner class*

```
public class Course implements Iterable<Student> {  
    private Student[] students;  
    ...  
    public Iterator<Student> iterator() {  
        return new StudentIterator();  
    }  
    private class StudentIterator  
        implements Iterator<Student> {  
        private int nextIndex;  
  
        public StudentIterator() {  
            nextIndex = 0;  
        }  
  
        public boolean hasNext() {  
            nextIndex < students.length;  
        }  
    }  
}
```

```
    public Student next() {
        Student s = students[nextIndex];
        nextIndex++;
        return s;
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}

...
Course vp = new Course("VP", "FEB23007", 2012, 3);
s.addStudent(new Student("tommi", 121212));
...
for (Student s : vp) {
    System.out.println(s.getName());
}
```

```
Iterator<Student> it = vp.iterator();  
  
while(it.hasNext()) {  
    System.out.println(it.next().getName());  
}
```

Loose vs tight coupling