Voortgezet Programmeren

Lecture 4: Interfaces and polymorphism

Tommi Tervonen

Econometric Institute, Erasmus University Rotterdam

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

- Interfaces are empty classes defining methods that have to be implemented by classes implementing the interface
- Interfaces declare types like normal classes do: when implementing an interface, a class can be of multiple types

```
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

- 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);
```

Dynamic binding

- All method implementations have to bound to its implementation (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*

Casting

- 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());
  }
}
```

How do you know your method works?

Unit testing

- Unit testing refers to automated testing of code functionality a "unit" at a time (e.g. method)
- We test only public methods (=the interface)
- It it's not tested, it doesn't work
- A single unit test tests one functionality, and tests can be grouped to test suites (usually we have 1 test suite with all tests)

```
public static void insSort(Keyable[] arr) { ... }
```

```
public class SorterTest {

  @Test
  public void testSort() {
    Student s1 = new Student("aa");
    Student s2 = new Student("bbb");
    Student s3 = new Student("c");
    Keyable[] elems = new Keyable[] { s1, s2, s2 };

    Sorter.sort(elems);

    assertTrue(elems[0] == s3);
    assertTrue(elems[1] == s1);
    assertTrue(elems[2] == s2);
  }
}
```

- Example: JSMAA-lib unit tests

```
public class StudentTest {
  private Student s;
  @Before
  public void setUp() {
    s = new Student("tommi", 1212);
  }
  @Test
  public void testConstructor() {
    assertEquals("tommi", s.getName());
    assertEquals(1212, s.getNumber());
  }
  @Test
  public void testSetGetName() {
    assertEquals("tommi", s.getName());
    s.setName("x");
    assertEquals("x", s.getName());
  }
  @Test
  public void testSetGetNumber() { ...
```

Why test?

- "If it's not tested, it doesn't work"
- Unit tests document functionality
- Unit tests provide a safety net ("let me change this ... does it break something?")
- More tests = more trust in your code
- Bug = **lack of a test**
- Test-driven development