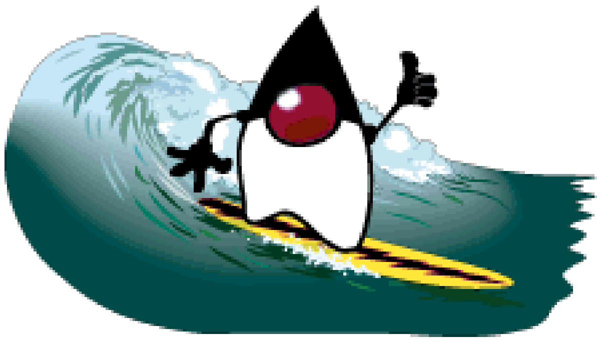
T5 - Java Seminar

T-JAV-500

Day 07

Generics



3.1

Day 07

language: Java

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

Let's dwelve deeper into OOP by studying one of Java's specificities: the generics.

Generics allow a class or a method to support multiple type while keeping the compile-time type safety. Without knowing it, you already used generics in day 01 by using the "ArrayList" type.

Let's have a look at the following block of Java code:

```java
List v = new ArrayList();
v.add("test");
Integer i = (Integer) v.get(0); // runtime error
```

Although the code compiles without error, it throws a runtime exception when executing the third line (*java.lang.ClassCastException*).

Take some time to understand this example! Test it on your own machine.

We would like this error to be detected at compilation time, and not at runtime, to avoid bad surprises... To cope with this matter, we need to use generics; in fact, this is the primary motivation for generics.

Exercise 01

File to hand in: ./Solo.java

Create a generic class named **Solo** that holds a value.
The class must have as attribute the value to hold, named *value*.
This attribute must have a getter (*getValue*) and a setter (*setValue*).

This class' constructor takes the value as parameter.

You probably need to make some research on Internet about generics' syntax...

We should be able to use your class strictly like in the following example:

```
Solo<String> strSolo = new Solo<>("toto");
String strValue = strSolo.getValue();
strSolo.setValue("tata");

Solo<Integer> intSolo = new Solo<>(Integer.valueOf(42));
Integer intValue = intSolo.getValue();
intSolo.setValue(Integer.valueOf(1337));
```

Exercise 02

File to hand in: ./Pair.java

Create a **Pair** generic class, that contains a pair of elements.
The types of both elements are a priori undefined.
The class must have the following attributes and one method:

- **first** is the first element of the pair, its type being mentioned as 'T',
- **second** is the second element of the pair, its type being mentioned as 'V',
- the **display** method that pretty prints the pair like this:

```
first: [first], second: [second].
```

The attributes need a getter (**getFirst**, **getSecond**) but no setter.
This class' constructor takes respectively the first and second element.

Exercise 03

Files to hand in: ./Duet.java

Create a **Duet** class that has two public static generic methods (that's long to say!) called respectively **min** and **max**.

These methods should take two parameters of T type and compare them using the **compareTo** method.
The **min** method returns the smallest one, whereas the **max** method returns the highest one.

smallest and *highest* can apply to many different types (not only numerical values), according to the **compareTo** method, which defines the ordering relation to be used.

To use the **compareTo** method, ensure that T extends from the Comparable interface.

Exercise 04

Files to hand in: ./Character.java
./Warrior.java
./Mage.java
./Movable.java
./Battalion.java
./Solo.java
./Pair.java
./Duet.java

Okay, now that you've started to understand how generics work, we're going to create battalions composed of Warrior and Mage (and potentially any other type of Character).

First, copy the *Character*, *Warrior* and *Mage* classes you wrote the previous day.
Then, create a **Battalion** class that has one attribute and two public methods:

- The attribute, of type List, must be called **characters**. It must hold all characters composing the battalion.
- A public method, **add**, which takes a List of Character objects (or any other object that inherits from Character) and add them to the battalion.

- A public method **display**, which displays the name of every character in *characters*.

For example:

```java
public static void main(String args[]) {
        List<Mage> mages = new ArrayList<>();
        mages.add(new Mage("Merlin"));
        mages.add(new Mage("Mandrake"));
        mages.add(new Mage("Adele"));
        List<Warrior> warriors = new ArrayList<>();
        warriors.add(new Warrior("Achilles"));
        warriors.add(new Warrior("Spartacus"));
        warriors.add(new Warrior("Clovis"));

        Battalion battalion = new Battalion();
        battalion.add(mages);
        battalion.add(warriors);
        battalion.display();
}
```

Terminal

```
~/T-JAV-500> java Example
Merlin: May the gods be with me.
Mandrake: May the gods be with me.
Adele: May the gods be with me.
Achilles: My name will go down in history!
Spartacus: My name will go down in history!
Clovis: My name will go down in history!
Merlin
Mandrake
Adele
Achilles
Spartacus
Clovis
```

Exercise 05

Files to hand in: ./Character.java
./Movable.java
./Warrior.java
./Mage.java
./Solo.java
./Pair.java
./Duet.java

Add a new integer attributes to the Character class: **capacity** that defaults to 0.
Add a new constructor that take the same parameters as the existing one but adding the capacity
For Warriors, this parameter represents it's strength.
For Mages, this parameter represents it's magnetism.
The Character class should also now implement the Comparable interface.
The implemented `compareTo` method will follow the following rules:

- If the argument is another character:
  - If both characters are Warriors, compare the strength,
  - If they are both Mages, compare the magnestism,
  - If there is a Warrior and a Mage, the Mage is the greatest unless the Warrior's strength is a multiple of the Mage's magnetism.
- Else just return 0

You need to be smart to compare a Mage and a Warrior...

For example:

```
Character merlin   = new Mage("Merlin", 12);
Character mandrake = new Mage("Mandrake", 9);
Character achilles = new Warrior("Achilles", 240);
merlin.compareTo(mandrake); // Should return 1
```

Exercise 06

Files to hand in: ./Character.java
./Movable.java
./Warrior.java
./Mage.java
./Battalion.java
./Solo.java
./Pair.java
./Duet.java

Add a **fight** method to the *Battalion* class.
It will take the first two characters of the battalion and make them battle against each other.
The winner will be determined using it's **compareTo** method.
The loser of the fight is then removed from the battalion (he is not worthy enough to stay... if he is still alive...).
The **fight** method will also return **true** if there is a fight or **false** is there is none (no characters in the battalion, ...).
In case of a tie, both characters are removed from the battalion (they're not worthy if they can't defeat their opponents).