Collections

More about collections

1. The *Collections* class

The Java Collections class contains static methods designed to operate with, or generate new collections. Those methods include generating synchronized collections (for safe operation with multiple threads), order a List (with some kind of comparator), reverse a List, rotate a List, get the minimum element, maximum, ...

2. Using generics in our own classes

When you look at Java's ArrayList reference web page, you'll see that this class is defined as `ArrayList<E>`. The symbol `<E>` is a notation used to define a generic class, a class that the compiler doesn't know until we instantiate an object. Another common notation is `<T>`, `<S>`, `<U>`, etc.. In general, a capital letter between `<` and `>`.

If we create a generic, by default it can be any class, so the only properties and methods that the compiler will allow us to use are the ones inherited from `Object` class (all classes in Java inherit from `Object`):

```
public class GenericExample<T>
{
    private T generic;

    public GenericExample(T generic)
    {
        this.generic = generic;
    }

    public void showType()
    {
        System.out.println(generic.getClass().getName().toString());
        // We can't use for example .substring()
        // since <T> can be anything.
    }

    public T getGeneric()
    {
        return generic;
    }
}
```

We define the type of `<T>` when we instantiate an object of GenericExample:

```
GenericExample<String> genEx = new GenericExample<>("Hello world!");
genEx.showType();   // java.lang.String

/* Here we can use a String method with
the generic object because the compiler
knows that the generic is a string */
System.out.println(genEx.getGeneric().length());
```

We can specify that the generic must be a subtype of class or implement some interface. In this case, as it happens with polymorphism, we can use the superclass or the interface's methods:

```
public class GenericExample<T extends Person>
{
    ...
    public void show()
    {
        // generic can use Person methods
        System.out.println(generic.getAge());
    }
    ...
}

// MAIN
GenericExample<String> genEx =
    new GenericExample<>("Hello world!");           // ERROR
GenericExample<Person> genEx =
    new GenericExample<>(new Person("Nacho", 40));  // OK
```

We can use more than one generic in a class:

```
public class GenericExample<T extends String, E extends Person>
{
    T attribute1;
    E attribute2;

    ...
}
```

2.1. An introductory example

Imagine we have a class called `Inventory` that can store items (any object that inherits from class `Item`). Let's see the difference in this case between using generics to define that class and using polymorphism.

The general appearance of `Item` class and some of its subclasses would be like this:

```
public class Item
{
    private float price;
    private int weight;

    public float getPrice()
    {
        return price;
    }

    public void setPrice(float price)
    {
        this.price = price;
    }

    public int getWeight()
    {
        return weight;
    }

    public void setWeight(int weight)
    {
        this.weight = weight;
    }
}

public class Potion extends Item
{
    public void drink()
    {
        System.out.println("Gulp gulp gulp.");
    }
}

public class Weapon extends Item
{
    private int damage;

    public int getDamage()
    {
        return damage;
    }

    public void setDamage(int damage)
    {
        this.damage = damage;
    }
}
```

If we use polymorphism to deal with a list of `Item` objects in our `Inventory` class, we would have something like this:

```
public class Inventory
{
    private List<Item> items = new ArrayList<>();

    public void addItem(Item item)
    {
        items.add(item);
    }

    public Item getItem(int index)
    {
        return items.size() > index?items.get(index):null;
    }
}

// MAIN
Inventory inv = new Inventory();
inv.addItem(new Potion()); // Index 0
inv.addItem(new Weapon()); // Index 1

// returns an Item, usually we don't know which type
Item it = inv.getItem(0);

if(it instanceof Potion)
{
    ((Potion) it).drink();
} else if(it instanceof Weapon) {
    System.out.println("Damage: " + ((Weapon) it).getDamage());
}
```

If we use generics to handle the same list, we would have this:

```
public class Inventory<T extends Item>
{
    private List<T> items = new ArrayList<>();

    public void addItem(T item)
    {
        items.add(item);
    }

    public T getItem(int index)
    {
        return items.size() > index?items.get(index):null;
    }
}

// MAIN
Inventory<Item> inv = new Inventory<>();      // Same behavior as before!
Inventory<Potion> potInv = new Inventory<>(); // Only allows potions
potInv.addItem(new Potion());                 // OK
potInv.addItem(new Weapon());                 // ERROR, <T> must be a Potion
Potion pot = potInv.getItem(0);               // Compiler knows is a Potion.
```

In summary, when we always want to be able to use more than one type of object inside a class instance, we can use polymorphism (an inventory with different types of items), although we still can use generics for this. When we want the possibility to use only one class at a time, defined when we instantiate an object, we can only do that with generics (we can create an inventory that only allows potions, other than only allows weapons and so on, using the same class for all).