More Object Orientation

and Git

Tobias Andersson Gidlund

tobias.andersson.gidlund@lnu.se

Agenda

Git

Abstract classes
- Abstract methods

Interfaces
- Your own interfaces
- Interfaces in Java

Introduction

- This lecture will cover two very important topics in object orientation:
  - Abstract classes
  - Interfaces
- Both help in minimising code errors by reuse and there encourage reuse.
- They can also be used with polymorphism to make it even more general.
- In addition, and to begin with, we will be looking at Git.

Git

Git in the course

- In this course we will introduce you to *git*, a distributed version control system.
- The main purpose is to store, track changes and share source code.
- Git is today an industry standard that is wildly used for version control.
- Several of you might already use it, but we will also introduce our own instance of GitLab in this lecture.
  - Otherwise the most common name among git is GitHub.
- This lecture will give you some background, some rules for using our GitLab and some guidelines for your workflow.

Background

- Version control is something that is needed when a project gets larger.
- On top of that, it also backups your files, which is a pro that many use it for even for smaller projects.
  - Manual version control could be to backup your files to folders on a cloud drive.
- During the years, several *Version Control Systems* or VCS have been developed and used.
  - CVS, Subversion and Visual SourceSafe to name a few.
- Many, if not most, so far have been *centralised*, which means that they have a server somewhere holding all the files.
- Git, any many like it, are *distributed*, meaning that everything (yes, everything) is stored locally.
  - But with the option to send it to a server as well for safe keeping.

Short, short history of Git

- Git is the brainchild of Linus Torvalds, developer of the Linux kernel.
  - Git means *unpleasant person* in British, which is something Linus sometimes feel like he is.
  - It could also mean “global information tracker”...
- It was developed in 2005 after the previously used DVCS BitKeeper revoked Linux’ free-of-charge license.
- It has evolved significantly since 2005, but the keywords *speed*, *simple design* and *fully distributed* still remain.
- Today it is more or less the defacto standard when using versioning control, both for open source software and commercial.

GitLab, GitHub and using git in class

- Git works as a “time machine” on your local machine, making it possible to keep track of changes (and revert if necessary).
- However, often we like to also make a backup somewhere and this is where sites like GitHub and GitLab comes in.
- GitHub is perhaps the most wildly used and know hosting company for git.
- At LNU we have opted for setting up our own instance of *GitLab* instead. `https://gitlab.lnu.se/`
- This is what we are going to use in our courses from now on.

GitLab at LNU

- Log in with your student account.
- Have a look at the Wiki which deals with most of the things you need to know.
- To get started, look at the dokumentation at `https://gitlab.lnu.se/instructions/get-started/tree/master`
  - Also worth mentioning is the instructions for multiple accounts.
- For each course you take, you will get a number of repositories for the tasks for that course.
  - This will be predefined for you, however, if the don't show up – contact us.

Basic workflow

- First follow the configuration steps linked to on the previous slide.
- To initiate the local git repository, go to the folder where you have your project and in the terminal and execute:

```
git init
```

- This will create a number of hidden files.
- To add a file to ignore your binary files and IDE (specify Intellij or Eclipse) specific files, use:

```
git ignore java,intellij >> .gitignore
```

- If the folder already has a number of files, the unhandled files and folders can be seen with:

```
git status
```

More basic workflow

- To add files to later be able to commit them, use:

```
git add [filenames]
```

- To commit the file, that is to say that it is done (for now) use:

```
git commit -m "Commit message"
```

- A `git status` should now say that everything is checked in.
- Next step is to push it to `https://gitlab.lnu.se/`
- First configure git to use our GitLab, see your project and use something like:

```
git remote add origin git@gitlab.lnu.se:tanmsi/gittutorial.git
```

- Push, that is send the code to the server, use:

```
git push -u origin master
```

Handing in assignments

- Moree information on `https://gitlab.lnu.se/instructions/get-started/blob/master/submit-an-assignment.md`
- Instead of using Moodle, we will now use GitLab.
- When you are done with an assignment (in time for the deadline) you:
  - Follow the instructions that will be up.
  - You will basically create and handle issues for your hand ins.
- The one correcting your assignment will modify the issue in GitLab.
- Make the corrections, add a message and *close* the issue to notify the teachers of the changes.

Working with Git

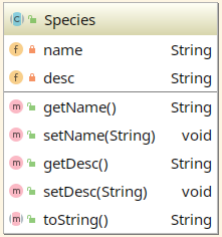
- Make frequent commits to your repo.
- It is not okay to make one large commit just before the deadline.
  - We will not impose a number of commits, but several are needed with some time between them.
- Make a habit of commiting (and pushing):
  - when you take a break
  - at the end of the day
  - when a task is done
- Failing to commit and push is not reason for failing the assignment alone, but will not be in your favour.

Abstract Classes

Abstract classes

- There are cases when it is not suitable to create an object from a class.
  - There might not be generic `Persons`, but rather either `Students` or `Teachers`.
- For this, it is possible to create an *abstract* class.
- This implies that the subclasses are the interesting parts, however they also have a common part.
- Abstract classes can have abstract methods – but also concrete methods.
  - An abstract method only has a signature, no body – much like an interface.

Example of an abstact class

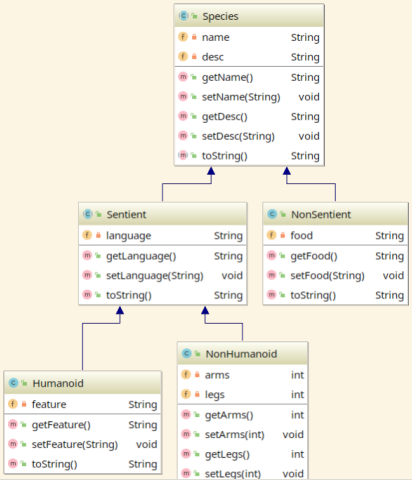



```java
public abstract class Species {
    private String name;
    private String desc;

    public Species() {
    }
    public Species(String name, String desc) {
        this.name = name;
        this.desc = desc;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDesc() {
        return desc;
    }

    public void setDesc(String desc) {
        this.desc = desc;
    }

    public abstract String toString();
}
```

More about abstract classes

- In the diagrams an abstract class is shown with an icon that looks like a tennis ball...
- Notice that as soon as a concrete subclass is created, it needs to implement the method.
- If there is at least one abstract method in a class, the entire class needs to be declared abstract.

The entire class diagram



The class `Sentient`

```
public abstract class Sentient extends Species {
    private String language;

    public Sentient() {
    }

    public Sentient(String name, String desc, String language) {
        super(name, desc);
        this.language = language;
    }

    public String getLanguage() {
        return language;
    }

    public void setLanguage(String language) {
        this.language = language;
    }

    public String toString() {
        return super.getName() + " with the description \"" + super.getDesc() + "\". " + getName() + " speaks " +
        ↪  language;
    }
}
```

The class NonSentient

```
public class NonSentient extends Species {
    private String food;

    public NonSentient() {
    }

    public NonSentient(String name, String desc, String food) {
        super(name, desc);
        this.food = food;
    }

    public String getFood() {
        return food;
    }

    public void setFood(String food) {
        this.food = food;
    }

    public String toString() {
        return super.getName() + " with the description \"" + super.getDesc() + "\". It eats " + food;
    }
}
```

The class `Humanoid`

```
public class Humanoid extends Sentient {
    private String feature;

    public Humanoid() {
    }

    public Humanoid(String name, String desc, String language, String feature) {
        super(name, desc, language);
        this.feature = feature;
    }

    public String getFeature() {
        return feature;
    }

    public void setFeature(String feature) {
        this.feature = feature;
    }

    public String toString() {
        return super.toString() + " and its feature is " + feature;
    }
}
```

The class NonHumanoid

```
public class NonHumanoid extends Sentient {
    private int arms;
    private int legs;

    public NonHumanoid() {
    }
    public NonHumanoid(String name, String desc, String language, int arms, int legs) {
        super(name, desc, language);
        this.arms = arms;
        this.legs = legs;
    }
    public int getArms() {
        return arms;
    }
    public void setArms(int arms) {
        this.arms = arms;
    }
    public int getLegs() {
        return legs;
    }
    public void setLegs(int legs) {
        this.legs = legs;
    }
    public String toString() {
        return super.toString() + " and has " + arms + " arms and " + legs + " legs";
    }
}
```

Main program

```
public class StarWarsCharacters {
    public static void main(String[] args) {
        Humanoid tiin = new Humanoid("Saesee Tiin", "was a male Iktochi from the moon Iktotch", "Basic", "horns");
        NonHumanoid jabba = new NonHumanoid("Jabba Desilijic Tiure", "was a Hutt and ganster", "Huttese", 2, 1);
        NonSentient tauntaun = new NonSentient("Tauntaun", "is a race of furry lizards from the planet Hoth", "floor
        ↪  lichen, ice scrabblers and Hoth hogs");

        System.out.println(tiin.toString());
        System.out.println(jabba.toString());
        System.out.println(tauntaun.toString());
    }
}
```

- Running:

```
Saesee Tiin with the description "was a male Iktochi from the moon Iktotch". Saesee Tiin speaks Basic and its feature is horn
Jabba Desilijic Tiure with the description "was a Hutt and ganster". Jabba Desilijic Tiure speaks Huttese and has 2 arms and
Tauntaun with the description "is a race of furry lizards from the planet Hoth". It eats floor lichen, ice scrabblers and Hot
```

Interfaces

Interface

- An interface is a contract that a class is promising to keep.
- It is similair to an abstract class except on one thing: it does *not* have an implementation.
  - This is why it is called a contract, it says what is *going* to be fulfilled.
- There are many interfaces available in Java, but it is also possible to create your own.
- In contrast to abstract classes it is possible to implement several interfaces in one class.
  - That is why it is called to *implement* rather than to extend an interface.
- Java 8 introduced something called *functionella interface* which is an interface with an implementation, but that can be disregarded for now.

Creating an interface

- An interface is created in the same way as a class, but use *Interface*.
- An interface may look like the following:

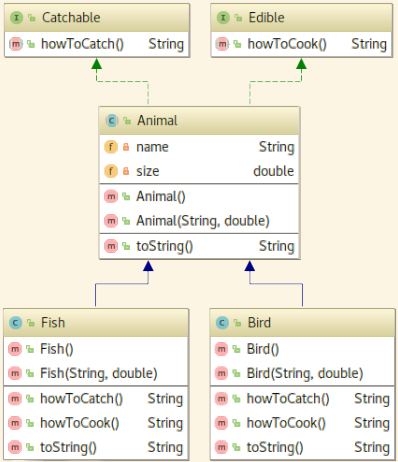
```
public interface Edible {
    public String howToCook();
}
```

- Classes that implement the contract may look like the following:

```
public abstract class Animal implements Edible, Catchable
```

- The above class is *abstract* and does not need to implement the methods, however the concrete classes inheriting from it must.
- It is practice to name it something ending with *-able*.

New example, UML diagram



The interfaces

- Each interface is put in its separate file.

```
package lecture8;

public interface Edible {
    public String howToCook();
}
```

```
package lecture8;

public interface Catchable {
    public String howToCatch();
}
```

Animal class

```
public abstract class Animal implements Edible, Catchable {
    private String name;
    private double size;

    public Animal() {
    }
    public Animal(String name, double size) {
        this.name = name;
        this.size = size;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getSize() {
        return size;
    }
    public void setSize(double size) {
        this.size = size;
    }
    public String toString() {
        return "A(n) " + name + " is " + size + " meters.";
    }
}
```

Fish class

```
package lecture8;

public class Fish extends Animal {
    public Fish() {
    }

    public Fish(String name, double size) {
        super(name, size);
    }

    public String howToCatch() {
        return "Catch using a fishing rod.";
    }

    public String howToCook() {
        return "Cook with loads of butter.";
    }

    public String toString() {
        return super.toString() + " " + howToCatch() + " " + howToCook();
    }
}
```

Bird class

```
package lecture8;

public class Bird extends Animal {
    public Bird() {
    }

    public Bird(String name, double size) {
        super(name, size);
    }

    public String howToCatch() {
        return "Catch by shooting.";
    }

    public String howToCook() {
        return "Barbecue it to eat.";
    }

    public String toString() {
        return super.toString() + " " + howToCatch() + " " + howToCook();
    }
}
```

A main program

```
package lecture8;

public class CatchingAndEating {
    public static void main(String[] args) {
        Fish aFish = new Fish("Nothern pike", 0.5);
        Bird aBird = new Bird("Ostrich", 2.5);

        System.out.println(aFish.toString());
        System.out.println(aBird.toString());
    }
}
```

Printout:

```
A(n) Nothern pike is 0.5 meters. Catch using a fishing rod. Cook with loads of butter.
A(n) Ostrich is 2.5 meters. Catch by shooting. Barbecue it to eat.
```

Predefined interfaces

- There are several interfaces in the Java API that a class can implement.
- The purpose is often to be able to use many of the algorithms that are available in the API that work for *everything* that implements them.
- One common example is `Comparable<T>`.
  - Instead of `<T>` you write the type (class).
- This interface has a method called `compareTo()` that returns a positive integer if the first object is larger than the second.

The class `Fish` with the interface `Comparable`

```
public class Fish extends Animal implements Comparable<Fish> {
    public Fish() {
    }
    public Fish(String name, double size) {
        super(name, size);
    }
    public String howToCatch() {
        return "Catch using a fishing rod.";
    }
    public String howToCook() {
        return "Cook with loads of butter.";
    }
    public String toString() {
        return super.toString() + " " + howToCatch() + " " + howToCook();
    }

    public int compareTo(Fish o) {
        if(this.getSize() > o.getSize()){
            return 1;
        }
        else if(this.getSize() < o.getSize()){
            return  -1;
        } else
            return 0;
    }
}
```

Main program

```
package lecture8;

public class ComparingFish {
    public static void main(String[] args) {
        Fish aFish = new Fish("Nothern pike", 0.5);
        Fish anotherFish = new Fish("European perch", 0.3);

        if(aFish.compareTo(anotherFish) > 0){
            System.out.println(aFish.getName() + " is the largest.");
        } else if(aFish.compareTo(anotherFish) < 0){
            System.out.println(anotherFish.getName() + " is the largest.");
        } else {
            System.out.println("They are equally large.");
        }
    }
}
```

Printout:

```
Nothern pike is the largest.
```

With `Arrays.sort()`

- The quick sort implemented in `Arrays.sort()` is defined to work for any class that implements `Comparable`.
- Therefore it is possible to create an array of fish and ask `Arrays.sort()` to sort it.
- In this case they will be sorted based on size.
  - By changing how `Comparable` is implemented in a class something else can be sorted by, for example the lexical order.
- Another interesting interface is `Cloneable` but it is left as an excersise.

Example sorting

```
import java.util.Arrays;
public class SortingFish {
    public static void main(String[] args) {
        Fish[] fishArray = new Fish[5];
        fishArray[0] = new Fish("Nothern pike", 0.5);     fishArray[3] = new Fish("Atlantic herring", 0.4);
        fishArray[1] = new Fish("European perch", 0.3);   fishArray[4] = new Fish("European flounder", 0.3);
        fishArray[2] = new Fish("Atlantic salmon", 4.5);

        for(Fish f: fishArray){
            System.out.print(f.getName() + " ");
        }
        System.out.println();

        Arrays.sort(fishArray);

        for(Fish f: fishArray){
            System.out.print(f.getName() + " ");
        }
    }
}
```

Printout:

```
Nothern pike European perch Atlantic salmon Atlantic herring European flounder
European perch European flounder Atlantic herring Nothern pike Atlantic salmon
```

Abstract class or interface?

- Both abstract classes and interfaces are useful, but sometimes difficult to choose between.
- When deciding, start by testing for “is-a”:
  - `Fish` *is-an* `Animal`? Yes, use abstract class.
  - `Fish` *is-a* `Comparable`? No, use interface.
- Apart from that, if no implementation is needed you will get cleaner code with interfaces.
- Also important to think of is if the sub classes need to be of several kinds.
  - Only interfaces are possible to use if several kinds are needed, only one class can be inherited from.
- Only classes can contain code, which sometimes is the deciding factor.

More on using interfaces and abstract classes

- Many of the predefined data structures in the Java Collection Framework (more later) use interfaces.
- This to make it possible to define a variable of an interface and later create the object itself.
- The following is possible as `ArrayList` is of the interface `List`:

```
List<String> list = new ArrayList<>();
```

- This works as all behaviour is specified in the interface (or, rather, what to expect) and the right hand provides an object.