Introduction and Inheritance

Moving on with Java

Tobias Andersson Gidlund

tobias.andersson.gidlund@lnu.se

Agenda

- Inheritance
- Inheritance in Java
  - Contructors
  - Overriding
- Polymophism
  - Substitution
  - Polymorphism
- Abstract classes
  - Abstract methods
- More on inheritance
  - instanceof
  - The Object class
  - Motivation and best practice

About the course

- Teachers: Jonas Lundberg (course responsible) and Tobias Andersson Gidlund
- Assistants:
  - Elise Anjel (English Group)
  - Ebtisam Mohammedsalih (English Group)
  - Björne Laanemäe (Swedish Group)
  - Robin Hägg (Swedish Group)
  - Madelene Amberman (Kalmar Group)
  - Fredrik Dahlberg (Distance)
- Notice that all *correcting* of assignments will be done by the Växjö assistants.
- Kalmar and distance assistents will still supervise their respective students.

More about the course...

- Exams:
  - 1st written exam March 22
  - 2nd written exam May 18
  - 3rd written exam in August...
- Content
  - Chapters 9 (rest) and 13-19 in Big Java – Late Objects
  - Chapters 11-27 excluding chapter 17 in Introduction to Java Programming by Liang
  - JavaFX (Liang or lecture notes)
  - Data structures (book)
  - Algorithms (book)
  - JUnit (other material)

Even more about the course...

- Goals for the course:
  - Have a working knowledge of *most* of Java.
  - You will be able to create programs with 10 – 15 classes.
- The main feature of Java *not* covered in the course is threads.
  - Used for creating concurrent programs.
- We will introduce several of the “modern” Java features, but not explore it in-depth.
  - Including lambdas, streams, JShell, modules, `var` keyword and so on.
- You will, however, be well prepared to look into any current and future development in the Java space!

Lectures

1. Introduction and Inheritance
2. Recursion and packages
3. New Java features
   Deadline Assignment 1
4. Basic Data Structures
5. JUnit testing
6. GUI (1)
   Deadline Assignment 2
7. Algorithms
8. Hashing + BST
9. GUI (2)
   Deadline Assignment 3
10. Generic classes and more algorithms
11. Preparations for written exam
    Deadline Assignment 4

Sessions

- Details can be found in TimeEdit (the schedule).
- Lectures:
  - Växjö: Monday 13-15 and/or Wednesday 10-12
  - Kalmar and distance: Tuesday 10-12 and/or Thursday 10-12
- Assignment sessions:
  - Växjö: Tuesday 10-12 and Thursday 15-17
  - Kalmar: Wednesday 14-16 and Friday 13-15
  - Distance: Skype, watch Moodle and Slack
- Assignments:
  - Four practical assignments.
  - All assignmnents are individual.
  - Each hand-in will be given a grade from A-F.
  - Read *assignment rules* in Moodle.

Admission and registration

- To attend this course, you need to be admitted.
  - If you are not – contact Admissions Office.
  - They will decide if you may take this course or not.
- Registration:
  - If you are admitted, you need to register!
  - First time students: Use online registration.
  - Retake students: Visit our Secretary Ewa Puschl.
  - Everyone needs to register (or reregister).
- Registration problems:
  - 1DV506 is a prerequisite for 1DV507 $\rightarrow$ most of you are not qualified!
  - You have done nothing wrong!
  - Pass any assignment in 1DV506 $\rightarrow$ you will be allowed to enter 1DV507

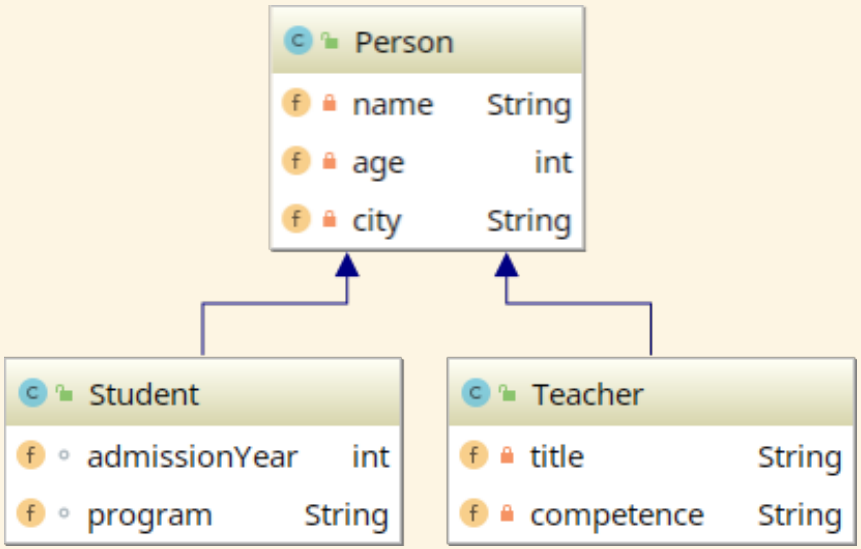
Inheritance

What it is

- This is chapter 9 in Horstmann or chapter 11 in Liang.
- Inheritance is a very powerful concept in object orientation.
- With inheritance a subclass can *extend* structure and behaviour of a super class.
- It is important to make the *is-a* test for every subclass.
  - Meaning saying that “student *is-a* person” and so on.
  - More on this later.
- Correctly used, inheritance makes programs more robust and easier to reuse.

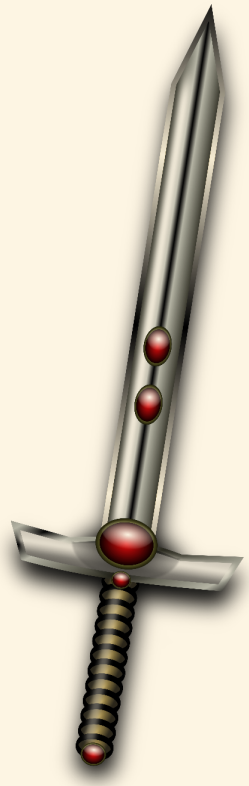
About inheritance

- Inheritance is a mechanism in object orientation that makes it possible for a class to attain the public parts of another class.
- It is said that the *subclass* gets all the public parts of a *super class* in a hierarchic structure.



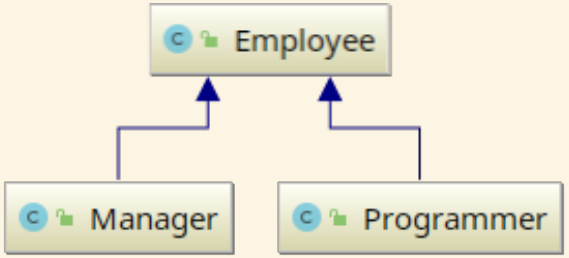
Inheritance – a double-edged sword

- The beginner tends to overuse inheritance, therefore there are a few things to consider.
  - This is the strongest relationship between two classes.
  - Encapsulation is weak for inheritance, changes in the super class are always propagated to the subclass – called “fragile base class problem”.
  - Inheritance is the least flexible relationship, for most languages it cannot be changed during runtime.
- It is of course not wrong to use inheritance, but it is important to understand it.



What can go wrong?

- Imagine the following structure:



- Say that there is a programmer in the company that gets “promoted” to a manager.
- What problems does that imply?

Discussion

- There is nothing wrong per se with the diagram and the inheritance.
- The problem arises when a programmer becomes a manager (or vice versa).
  - When this happens, a new object (a manager) needs to be created.
  - All information in the programmer object needs be copied to the new object.
  - Lastly, the programmer object has to be deleted.
- The problem is that there is a mismatch between reality and the system.
  - No new employee has been hired...
- So, it is important to as questions like: *are* you your job or do you *have* a job?

Inheritance in Java

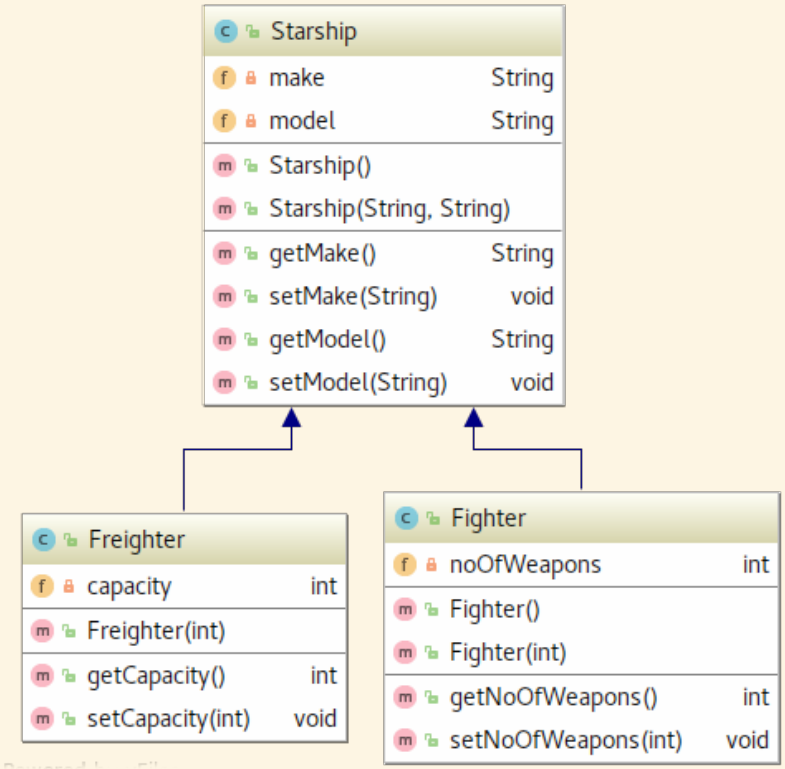
Extending

- Inheritance means that the super class is expanding with the content of the subclass.
- The keyword in Java is `extends`.

```
public class Fighter extends Starship {
  // additional code
}
```

- In this case there must exist a class called `Starship` which `Fighter` will expand upon.
- With Java it is only possible to inherit from one direct super class, but the height can be greater.
  - For example `Person` → `Employee` → `Programmer` → `JavaProgrammer`

Exemple




```java
package lecture8;

public class Starship {
    private String make;
    private String model;

    public Starship() {
    }
    public Starship(String make, String model) {
        this.make = make;
        this.model = model;
    }

    public String getMake() {
        return make;
    }

    public void setMake(String make) {
        this.make = make;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }
}
```

Subclasses

```java
package lecture8;

public class Fighter extends Starship {
    private int noOfWeapons;

    public Fighter() {
    }

    public Fighter(int noOfWeapons){
        this.noOfWeapons = noOfWeapons;
    }

    public int getNoOfWeapons() {
        return noOfWeapons;
    }

    public void setNoOfWeapons(int noOfWeapons) {
        this.noOfWeapons = noOfWeapons;
    }
}
```

```java
package lecture8;

public class Freighter extends Starship {
    private int capacity;

    public Freighter(int capacity) {
        this.capacity = capacity;
    }

    public int getCapacity() {
        return capacity;
    }

    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }
}
```

Creating objects

- It is then possible to create objects of the subclasses just like previously.
  - And, as it is right now, also from the super class.
- Objects are created using the `new` keyword as always:

```
Fighter xwing = new Fighter();
```

- The object has *all* the parts of the super class making it possible to do the following:

```
xwing.setMake("Incom Corporation");
```

- It is, of course, also possible to set the values of attributes defined in the subclass:

```
xwing.setNoOfWeapons(6);
```

Complete example

```
public class Hangar {
    public static void main(String[] args) {
        Fighter xwing = new Fighter();
        xwing.setMake("Incom Corporation");
        xwing.setModel("T-65 X-wing starfigher");
        xwing.setNoOfWeapons(6);

        System.out.println("I fly a " + xwing.getModel());
        System.out.println("from " + xwing.getMake());
        System.out.println("and fire with my " + xwing.getNoOfWeapons()
                + " weapons!");
    }
}
```

- When running:

```
I fly a T-65 X-wing starfigher
from Incom Corporation
and fire with my 6 weapons!
```

Constructors and inheritance

- Even if constructors are public, they are *not* inherited to the subclass.
- Just as previously, if no constructor is defined, there is a default constructor.
- It is then possible to replace it with one or more constructors.
- There are two options for setting values:
  - Using public methods.
  - Using `super()` as a reference to the superclass.
- `super()` can be used to call both constructors and methods in the super class.

Updated `Fighter`

```
public class Fighter extends Starship {
    private int noOfWeapons;

    public Fighter() {
    }

    public Fighter(String make, String model, int noOfWeapons) { // new constructor
        super(make, model);
        this.noOfWeapons = noOfWeapons;
    }

    public Fighter(int noOfWeapons){
        this.noOfWeapons = noOfWeapons;
    }

    public int getNoOfWeapons() {
        return noOfWeapons;
    }

    public void setNoOfWeapons(int noOfWeapons) {
        this.noOfWeapons = noOfWeapons;
    }
}
```

Overriding methods

- Methods from super classes can be overridden in the subclass.
- This means that for a method in the super class, there is an exact dublicate in the subclass.
  - Same signature, but not the same body.
  - The subclass will *redefine* the behaviour.
- It is possible for the subclass to call the super class using `super`.
  - This is useful if the super class need to define a number of variables.
- The following example will show how to override `toString()` that is actually from the class `Object`.

Changes

- In `Starship` we add the following method:

```
public String toString() {
    return "This is a " + make + " " + model;
}
```

- In `Fighter` this is overridden using the following:

```
public String toString() {
    return super.toString() + " with " + noOfWeapons + "
    ↪  weapons.";
}
```

- In main the following can be executed:

```
System.out.println(xwing.toString());
```

- Which will display:

```
This is a Incom Corporation T-65 X-wing starfigher with 6 weapons.
```

Overriding and overloading

- Notice that there is a difference between *overloading* as done before and *overriding*.
  - Overloading is about having several methods with the same name but different signatures.
  - Overriding is about overwriting the behaviour that already exists.
- In the next step, we will look at *polymorphism* and in that case overriding will be very important.
- Also notice that it is possible to prevent a method from being overriden by declaring it as `final`.

Polymorphism

Substitution

- An advantage with inheritance is that all subclasses can be instatiated for a super class.
  - We do not need to know if it will be a `Fighter` or a `Frieghter`, just that it will be a `Starship`.
- This is possible since subclasses always are *types-of* the super class.
- The following example will create a list of `Starships` and be able to put any kind of subclass in it.
- At runtime, one of the methods in the super class will be called for each object.
- This is what is called *substitution* in programming.

Example

```
public class FullHangar {
    public static void main(String[] args) {
        Starship[] listOfShips = new Starship[3];
        listOfShips[0] = new Fighter("Incom Corporation", "T-65 X-wing starfigher", 6);
        listOfShips[1] = new Freighter("Corellian Engineering Coporation", "YT-1300", 100);
        listOfShips[2] = new Fighter("Koensayr Manufacturing", "BTL Y-wing", 5);

        for(Starship s: listOfShips) {
            System.out.println("The model is called " + s.getModel());
        }
    }
}
```

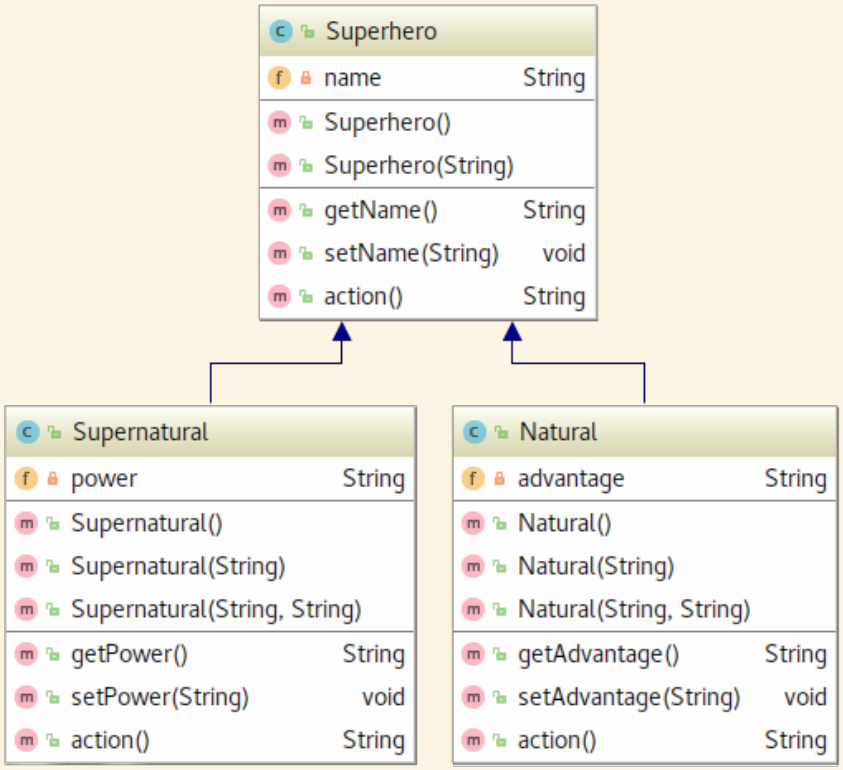
- When running:

```
The model is called T-65 X-wing starfigher
The model is called YT-1300
The model is called BTL Y-wing
```

Polymorphism

- The word “polymorphism” is from greek and roughly translates to *many forms*.
- It is an important part of object orientation and often seen as a key feature.
- In essence it is about allowing for a method to be redefined in a subclass.
- The difference from the previous examples is that it the correct method will be selected *at runtime* rather than *compile time*.
- This will make the programs much more flexible.
- This is also called *dynamic binding* to the correct method.

Example




```java
public class Superhero {
    private String name;

    public Superhero(){
    }

    public Superhero(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String action() {
        return name + " will ";
    }
}
```

Example, cont.

```java
public class Supernatural extends Superhero {
    private String power;

    public Supernatural() {
    }

    public Supernatural(String power) {
        this.power = power;
    }

    public Supernatural(String name, String power) {
        super(name);
        this.power = power;
    }

    public String getPower() {
        return power;
    }

    public void setPower(String power) {
        this.power = power;
    }

    public String action() {
        return super.action() + power
                + " to save you!";
    }
}
```

```java
public class Natural extends Superhero {
    private String advantage;

    public Natural(){
    }

    public Natural(String advantage) {
        this.advantage = advantage;
    }

    public Natural(String name, String advantage) {
        super(name);
        this.advantage = advantage;
    }

    public String getAdvantage() {
        return advantage;
    }

    public void setAdvantage(String advantage) {
        this.advantage = advantage;
    }

    public String action() {
        return super.action() + "use "
                + advantage + " to save you!";
    }
}
```

Main program

```
public class Heroes {
    public static void main(String[] args) {
        Superhero[] myHeroes = new Superhero[5];

        myHeroes[0] = new Supernatural("Superman", "fly");
        myHeroes[1] = new Natural("Iron man", "superiour technology");
        myHeroes[2] = new Natural("Batman", "his fantastic brain");
        myHeroes[3] = new Supernatural("Wolverine", "fight with claws");
        myHeroes[4] = new Supernatural("The Flash", "fun fast");

        for(Superhero s: myHeroes){
            System.out.println(s.action());
        }
    }
}
```

Running:

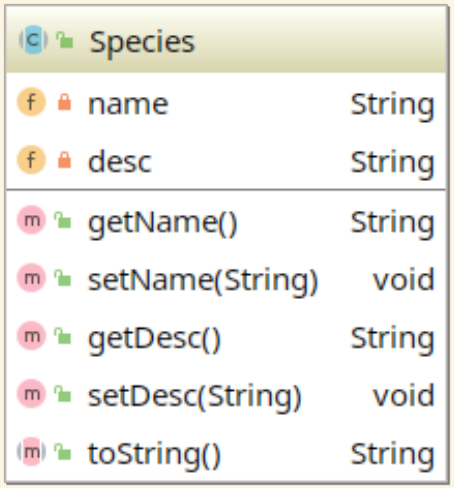
```
Superman will fly to save you!
Iron man will use superiour technology to save you!
Batman will use his fantastic brain to save you!
Wolverine will fight with claws to save you!
The Flash will fun fast to save you!
```

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

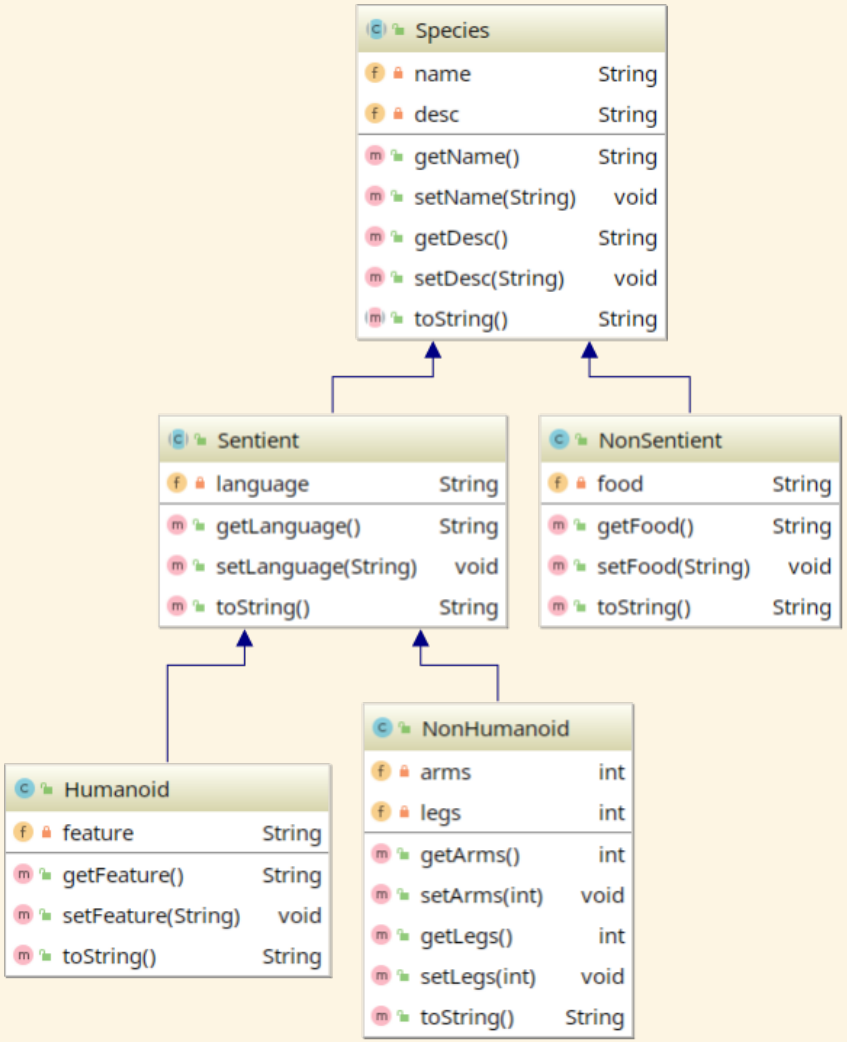



```java
public abstract class Species {
    private String name;
    private String desc;

    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...
- Just as with interfaces, it is important to include the semicolon for abstract methods.
- In contrast to interfaces, an abstract class can mix both concrete and abstract methods.
- 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 B
Jabba Desilijic Tiure with the description "was a Hutt and ganster". Jabba Desilijic Tiure speaks
Tauntaun with the description "is a race of furry lizards from the planet Hoth". It eats floor li
```

More On Inheritance

`instanceof` operator

- To find out the specific instance of a class, it is possible to use the `instanceof` operator.
  - In the Java all keywords must be in lower case, that is why it's not `instanceOf`...
- As it is a binary operator it has a left hand and a right hand and it returns true or false.
  - The left hand is the object to test.
  - The right hand is a class.
- Notice that classes in a hierarchy always return true for all super classes, but not for subclasses.

Example

```
public class CheckHangar {
    public static void main(String[] args) {
        Starship xwing = new Fighter();
        Starship aShip = new Starship();

        if(xwing instanceof Fighter)
            System.out.println("It\'s a fighter.");

        if(xwing instanceof Starship)
            System.out.println("It\'s a starship.");

        if(!(aShip instanceof Fighter))
            System.out.println("It\'s not a fighter.");
    }
}
```

- Output:

```
It's a fighter.
It's a starship.
It's not a fighter.
```

The class `Object`

- There is one special class in the Java class hierarchy – `Object`.
- This class is the root of *all* classes in the hierarchy.
- All classes, both those in the hierarchy and those created by us, inherit implicitly from `Object`.

```
public class Starfigher {  <=> public class Starfigher extends Object {
 ...                                ...
}                                  }
```

- One of the reasons for having such a class is to have methods that are available in all classes.

Usage

- Due to the inheritance, *all* classes conform to the “is-a” `Object` test.
- Before generics was added to Java in 1.5 most datastructures used `Object` as the datatype stored.
- Today the best usage is to override `Object`’s methods:
  - `boolean equals( Object o );`
  - `Class getClass();`
  - `int hashCode();`
  - `void notify();`
  - `void notifyAll();`
  - `String toString();`
  - `void wait();`
  - `protected Object clone() throws CloneNotSupportedException;`
  - `protected void finalize() throws Throwable;`

Some of the “more important” methods...

- Some of the methods are rarely used unless for use in threads.
- However, a handfull are often used.
- Already covered is the `toString()` method that returns a string representation of the object.
  - Used previously to create a string for each starfigher or species.
- The `equals(Object o)` method is used to determine if two objects are the same.
  - The default implementation in `Object` checks the allocated memory place.
- `getClass()` is useful to find out the class for an object.
- Later in the course you will also implement `hashCode()`.

Example of default implementation

```
public class JediObjectCheck {
    public static void main(String[] args) {
        Jedi tiin = new Jedi("Saesee Tiin");
        Jedi saesee = new Jedi("Saesee Tiin");

        System.out.println(tiin.toString());
        System.out.println(saesee.toString());
        System.out.println(tiin.getClass());

        if(tiin.equals(saesee))
            System.out.println("The same");
        else
            System.out.println("Not the same"
);
    }
}
```

- Output:

```
Lecture1.Jedi@1b0375b3
Lecture1.Jedi@2f7c7260
class Lecture1.Jedi
Not the same
```

```
public class Jedi {
    private String name;

    public Jedi() {
    }

    public Jedi(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Updated `Jedi` class

```
public class Jedi {
    private String name;

    public Jedi() {
    }
    public Jedi(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String toString() {
        return name;
    }

    public boolean equals(Object obj) {
        if(obj instanceof Jedi) {
            if(this.name.equals(((Jedi) obj).getName()))
                return true;
        }
        return false;
    }
}
```

- Output of the same main program:

```
Saesee Tiin
Saesee Tiin
class Lecture1.Jedi
The same
```

Motivation

- One of the promises of object orientation was easy code reuse and inheritance is part of that.
  - Code from the super class is reused in subclasses, which means less code writing and less places to correct errors.
  - Behaviour is also reused – or explicitly overwritten by the subclass.
- Inheritance also gives an easier to understand and maintain organisation of the code base.
- As subclasses can be created from other classes, inheritance makes the program extendable.
  - If there already exists a class `Circle` it is easy to extend it to `Ellipse`.

Best practice

- Find classes which holds common features and make them super classes.
  - Still important to test for “is-a”!
- Make super classes abstract as often as possible – do we really need objects of it?
- Always try to move behaviour upwards in the hierarchy.
- Java has *annotations*, the possibility to add `@Override` for methods that are overridden – use that.
  - Annotations are instructions to the compiler to verify that existing methods are overridden.
  - Often neglected in lecture slides due to space...