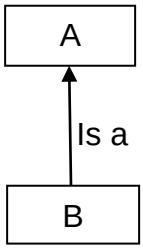
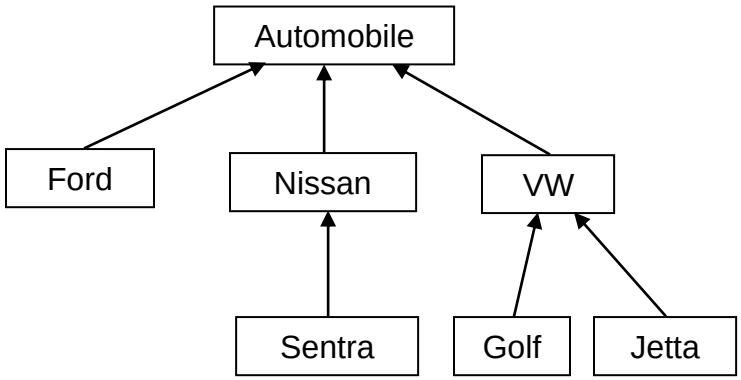
Advanced Object Oriented Programming

Introduction to Inheritance: Extending a Class

- Inheritance enables us to define a new class based upon an existing class. The new class is similar to the existing class, but has additional member variables and methods. For example, a new class `Student` can be defined based on an existing class `Person`.
- In diagrams that show inheritance, an arrow points from the new class to the class it is based upon. The arrow is sometimes labeled "is a"
- In this diagram, class `B` is a `A`. Class `B` is based upon class `A`. Class `A` is the **superclass** (or parent class or base class), and class `B` is the **subclass** (or child class or derived class).



- In Java, each class can inherit characteristics from just one superclass.
- Inheritance is between classes, not objects.
- A class hierarchy demonstrates the relationships among classes. In a hierarchy, each has at most one parent but might have several children classes.



- To define a class based on another class, the keyword `extends` is used:

```
class childClass extends parentClass {
     // new characteristics of the child class go here
}
```

e.g.

```
class Person {
     String name;
     char gender;
     int age;
}

class Student extends Person {
```

```
        int studentNum;
        int grade;
    }
```

The Student class has the instance fields of `studentNum` and `grade` as well as `name`, `gender` and `age`.

- If no superclass is specified by an `extends` clause, `Object` is the default superclass. Therefore, all classes inherit from `Object` – either directly or indirectly by extending some other class that inherits from `Object`.

Using `super`

- The superclass of a class can be referred to using the reserved word `super`. This can be used to invoke a constructor of the superclass with some arguments. The call to the superclass' constructor must be the very first statement in a constructor.
  e.g.

```
class Student extends Person {
        private int studentNum;
        private int grade;

        public Student (String name, char gender,
                int age, int studentNum, int grade) {
                super(name, gender, age);
                this.studentNum = studentNum;
                this.grade = grade;
        }
}
```

- Even without an explicit call of the superclass constructor, the statement `super()` is inserted automatically as the first statement of the constructor. This ensures that in constructing an object, any inherited fields in initialized.
- `super` can be used to call methods of the superclass.
  e.g. In the `Person` class, the `toString` method can be defined as followed:

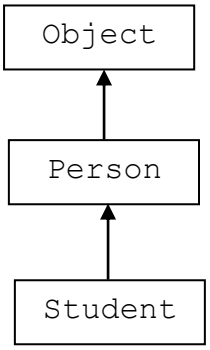
```
        public String toString() {
                return "Name:" + name + "\ngender:" + gender
                + "\nage:" + age + "\n";
        }
```

Then the `toString` method in the `Student` class can make use of the `toString` method in the `Person` class:

```
        public String toString() {
                return super.toString() +
                " #:" + number + "\ngrade:" + grade;
        }
```

- `super` can also be used to get access to the shadowed fields or overridden method in a superclass.

Polymorphism: Inheritance and Variables



- Polymorphism is an OOP property in which objects have the ability to assume different types. It is based on inheritance.
- Since a subclass is derived from a superclass, a superclass object can reference an object of the subclass:

```
Student s = new Student();
Person p = new Student();
Object o = new Student();
```

- We can never assign an object to a reference variable of another type unless the two types are in the same hierarchy.
- Within the same hierarchy, any assignment upward is valid, but not the other way around. Therefore, the following assignment is invalid:

```
Object o = new Object();
Student s = o;
```

- If the assignment is made to a more restrictive type of object (further down in the hierarchy), casting must be used. This is sometimes called downcasting because the target is lower in the hierarchy.
  e.g.

```
Object o = new Student();
Student s = (Student)o;
```

- It is important to make sure that the cast make sense.
  e.g.

```
Object o = new Object();
Student s = (Student)o;
```

  The above fragment compiles, however when the code is executed, it will throw a `ClassCastException` error because `o` (of type `Object`) does not have a `Student` part.

Accessibility of Variables

- Any fields declared with `private` visibility modifier can only be seen from within their own class, but not anywhere else, not even the subclass within same hierarchy (accessor and mutator methods must be used)
- The `protected` visibility modifier gives subclasses direct access to fields. Any fields declared with the `protected` attribute can be seen either in that class or any of its subclasses but not elsewhere.
  e.g.

```
class Sample {
     public int a;
     float b;
     protected char c;
```

```
    private boolean d;
}
```

  - `a` is visible everywhere
  - `b` is visible in the package in which `Sample` is defined
  - `c` is visible within `Sample` and any of its subclasses
  - `d` is visible with `Sample`
- A field declared in a subclass can have the same identifier as a field in its superclass. In this case, the field in the superclass is **shadowed** by the one in the subclass and it is no longer visible in the subclass.