

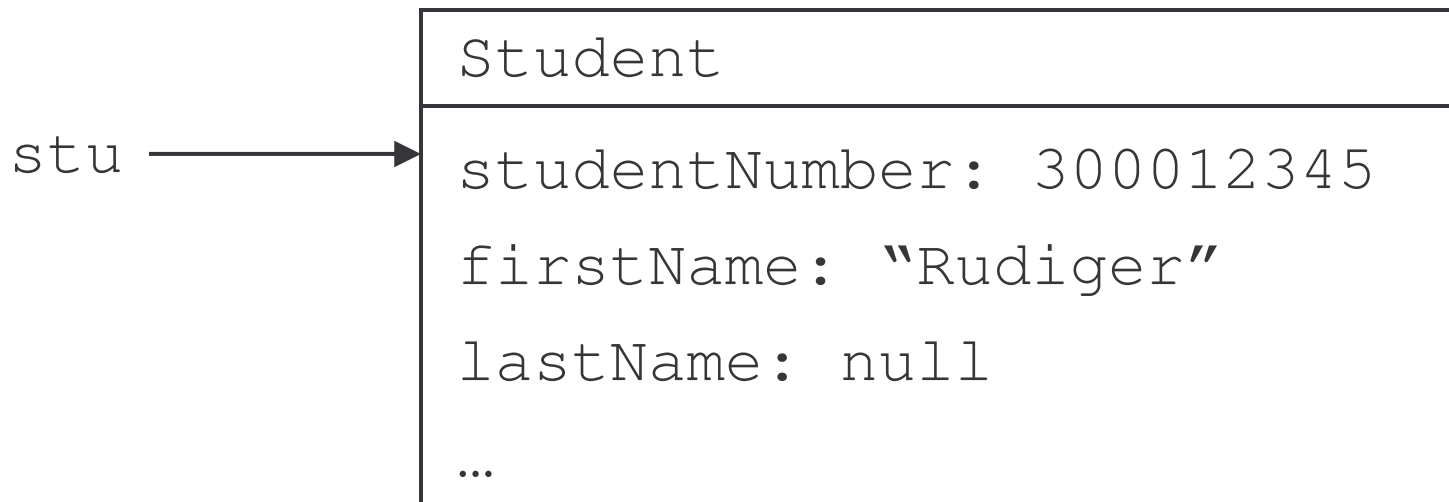
# Working with References

# References

- Every object variable in Java is really a **reference** to an object.
- Also true when an object is passed as an argument: a **reference** to the object is passed to the function.
- When the object is used, the reference is “followed” to find the actual object.

# The Picture

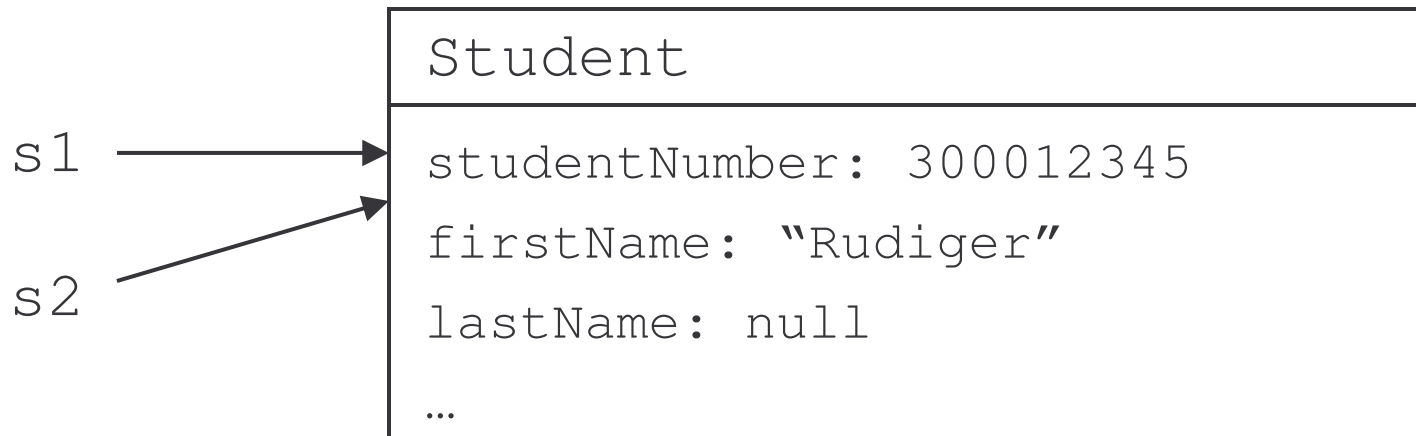
```
Student stu = new Student(300012345, "uid");  
stu.setFirstName("Rudiger");
```



# Aliases

- Assigning to an object copies the **reference**, not the whole object:

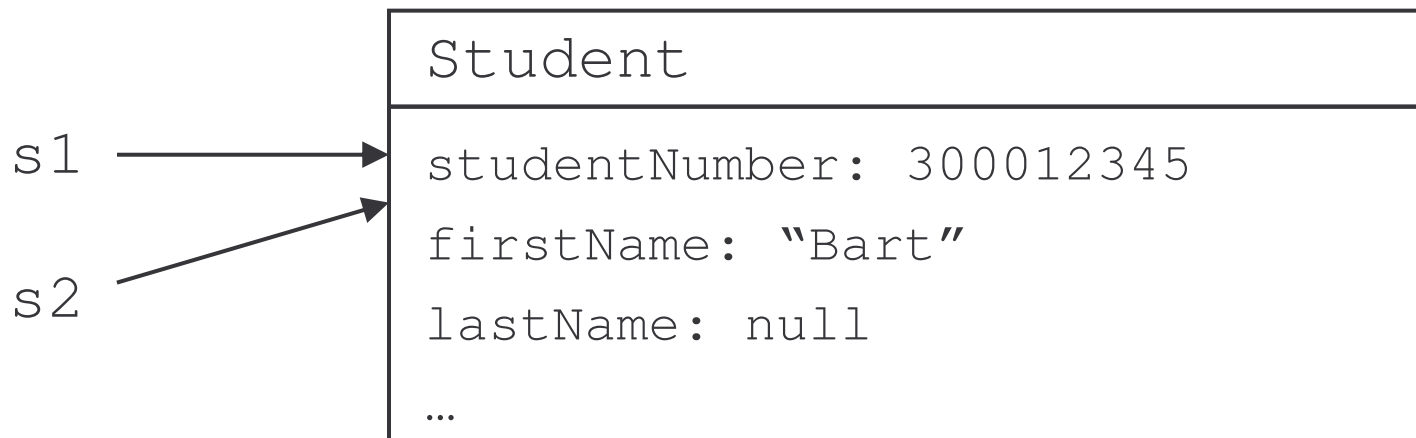
```
Student s1 = new Student(300012345, "uid");  
s1.setFirstName("Rudiger");  
Student s2 = s1;
```



# Aliases

- Then, changes to one object will affect both references:

```
s2.setFirstName("Bart");  
System.out.print( s1.getFirstName() );  
// prints "Bart"
```



# Copying

- If we really do want to **copy** an object, it has to be done manually.
- Create a new instance and copy the relevant data over.
- Not an issue if there are no methods that modify the object: no unexpected results from changes.
  - eg. the `String` class
  - called “immutable objects” in Python

# The `clone` Method

- Many classes contain a `.clone()` method.
  - This method returns a copy of the object.
  - i.e. a **new** object with the relevant data copied
- eg. 

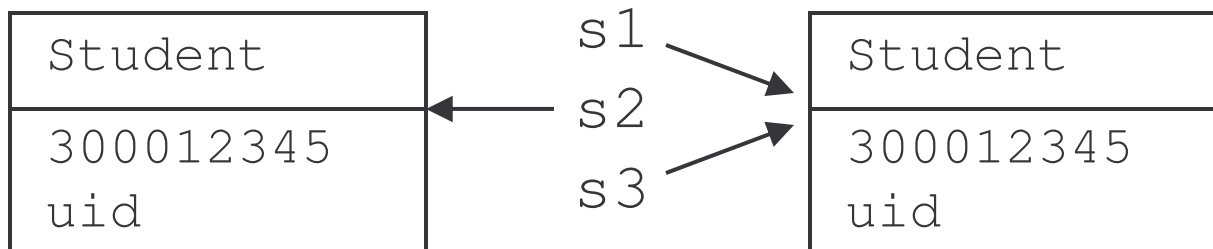
```
public Student clone() {  
    Student s = new Student(  
        studentNumber, userid);  
    ... // copy rest of the data to s  
    return s;  
}
```

# Equality and References

- When comparing two objects, the **references** are compared, not the object contents.

```
Student s1 = new Student(300012345, "uid");  
Student s2 = new Student(300012345, "uid");  
Student s3 = s1;
```

- Now: `s1==s3 && s1!=s2`






# The equals Method

- Many classes contain a `.equals()` method.
  - Takes another object of the same type as its argument.
  - Return true if the two objects are “equal”
  - ... whatever “equal” means for this type.

■ eg.

```
public boolean equals(Student s) {  
    return studentNumber==s.studentNumber;  
}
```

The diagram shows two arrows pointing upwards from labels below to variables in the code above. One arrow points from 'this instance's data member' to 'studentNumber' in the return statement. The other arrow points from 'other instance's data member' to 's.studentNumber' in the return statement.

this instance's data member

other instance's data member

# The `compareTo` Method

- Used for more general comparison: `<`, `==`, `>`
- `a.compareTo(b)` should return:
  - a negative `int` if `a < b`
  - `0` if `a==b`
  - a positive `int` if `a > b`
- Used by the built-in sorts for objects.
  - One call to `compareTo` gives all the info needed about relative order.

# The “this” Reference

- It is often convenient/necessary to explicitly refer to members of the current object.
  - eg. `return studentNumber==other.studentNumber;`
  - not totally clear where the first `studentNumber` comes from.
- There is a special variable `this` that always refers to the object that the code is defining.
  - previous example is now easier to read:  
`return this.studentNumber==other.studentNumber;`

# Specifying Scope with **this**

- In methods, we had to be careful to choose different names for arguments and data members.

```
public Student(long stunum) {  
    studentNumber = stunum; }  
}
```

- Using `this`, we can avoid the problem & keep names consistent:

```
public Student(long studentNumber) {  
    this.studentNumber = studentNumber; }  
}
```