

Unit 02:

Classes and Objects

Anthony Estey

CSC 115: Fundamentals of Programming II

University of Victoria

Unit 01 Overview

- ▶ Related Reading:

- ▶ Textbook pages 23-25

- ▶ Learning Objectives: (You should be able to...)

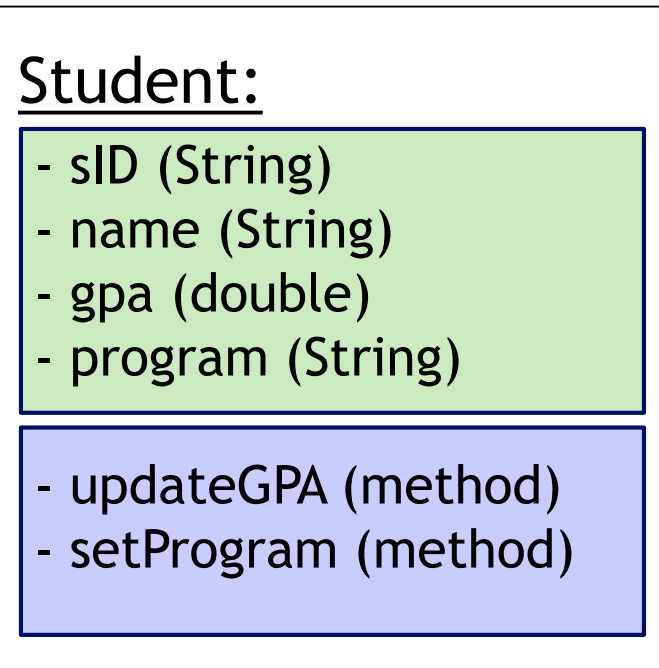
- ▶ create and use your own objects and classes in Java
 - ▶ describe what it means to create an instance of a class
 - ▶ describe the purpose of the following:
 - ▶ data fields
 - ▶ constructors
 - ▶ setters and getters (or accessors and mutators)
 - ▶ toString
 - ▶ describe the difference between non-static and static methods

Classes and Objects

- ▶ So far the variables we have created have allowed us to represent numbers and text...
- ▶ ... but sometimes the information we work with in our programs cannot be adequately represented by a single number or text
- ▶ Example:
 - ▶ We could represent a song with simply a title...
 - ▶ ... but we may also want to know about the artist, duration, genre, etc.
 - ▶ Similarly, we could represent a student by a student number...
 - ▶ ... but we may also want to know their name, program, gpa, etc.

Classes

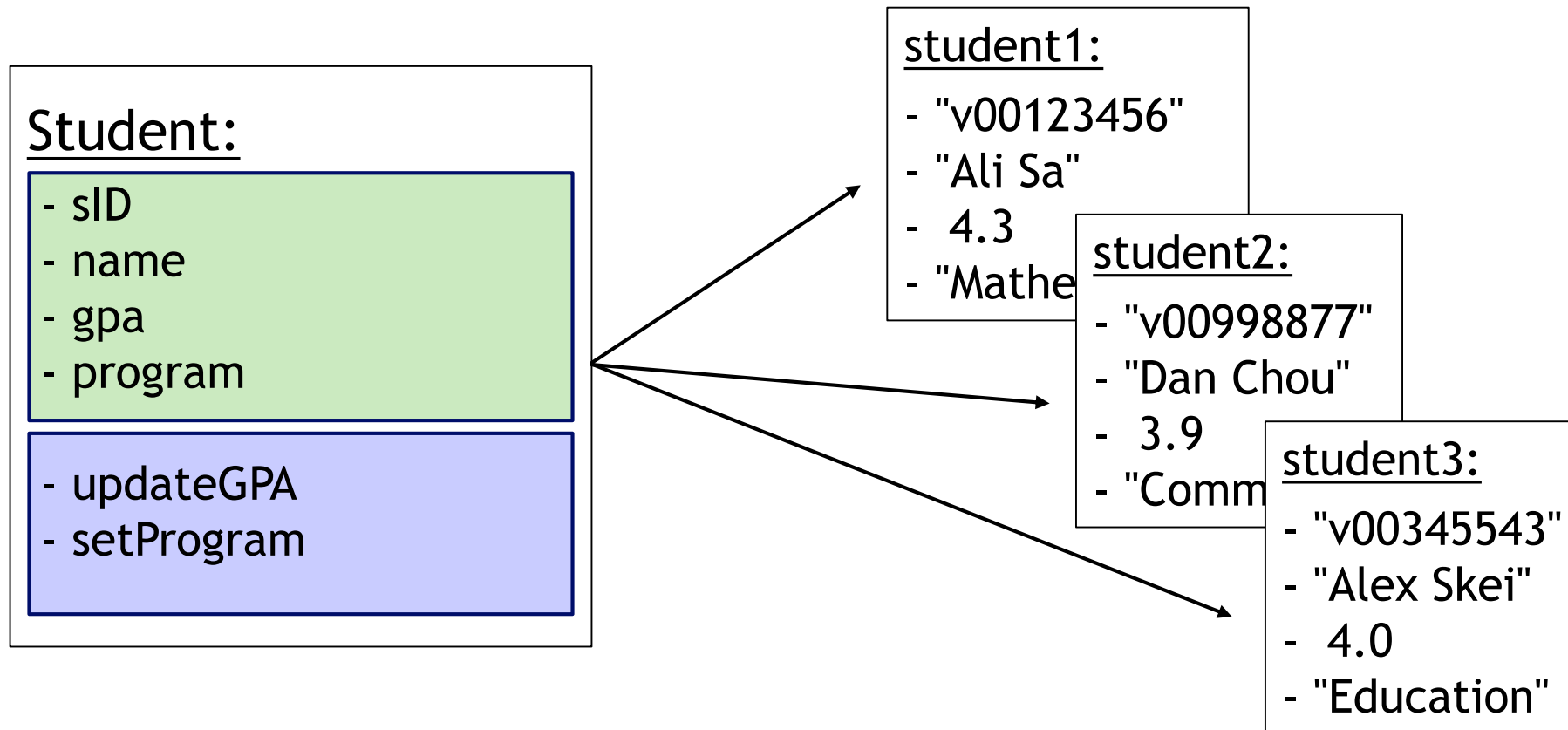
- ▶ Classes allow us to define our own data types by defining:
 - ▶ their **attributes** (that help us differentiate one from another)
 - ▶ and **associated behaviours** (what operations we can do with it)
- ▶ Student example:



We can think of a class as the blueprint for an object

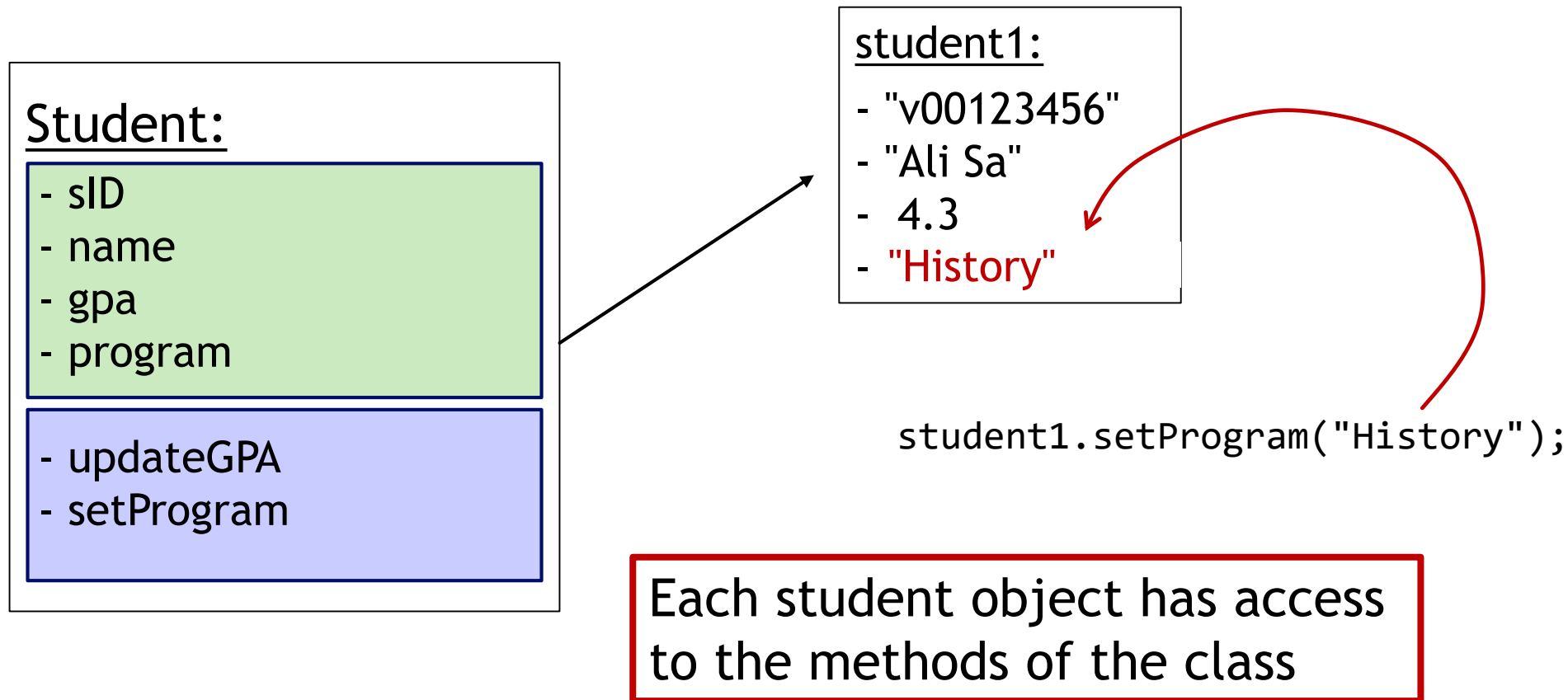
Classes and Objects

- ▶ We consider a class the blueprint for an object...
- ▶ ... when we create an object we say that it is an *instance* of a class



Classes and Objects

- ▶ We consider a class the blueprint for an object...
- ▶ ... when we create an object we say that it is an *instance* of a class



Fields

- ▶ Another name for a class attributes is a **field**
- ▶ **Fields** can be accessed and updated like variables:
 - ▶ output: `System.out.println(fieldname);`
 - ▶ modify: `fieldName = newValue;`
- ▶ For example:
 - ▶ The Student class has four fields
 - ▶ The student1 object is an instance of the Student class
 - ▶ The diagram on the right illustrates the values for each field

student1:

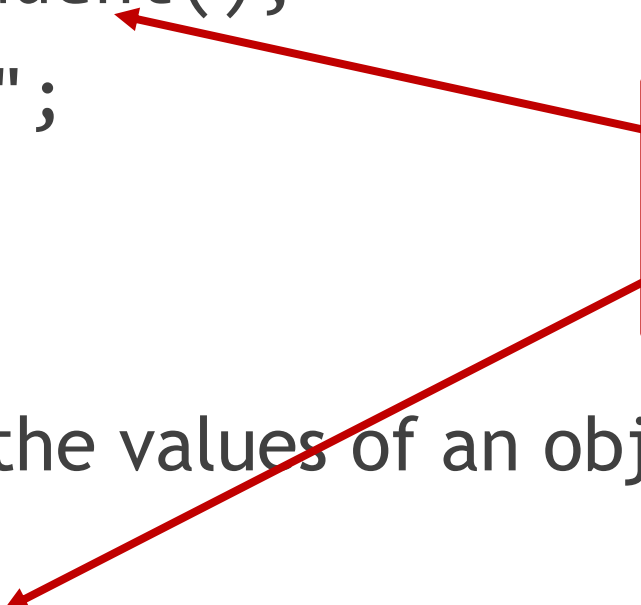
- sID: "v00123456"
- name: "Ali Sa"
- gpa: 4.3
- program: "Mathematics"

Initializing Objects

- ▶ **new**: the new keyword allocates memory for a new instance of a class:

```
Student s1 = new Student();  
s1.sID = "v00123456";  
s1.name = "Ali Sa";
```

These lines of code
both call the Student
class' **Constructor**



- ▶ There is a way to initialize the values of an object's fields when it is first declared:

```
Student s1 = new Student("v00123456", "Ali Sa", ...);
```

Constructor

- ▶ The **constructor** initializes the **field** values when an object is created
- ▶ The **constructor** is automatically called when the `new` keyword is used
- ▶ General syntax:

```
public Name(parameters) {  
    statements;  
}
```

- ▶ A constructor looks similar to a method, with some key differences:
 - ▶ the name of the constructor *always* matches the class name
 - ▶ no return type is specified
 - ▶ constructors implicitly return the new object being created

Constructor

- ▶ If a class has no constructor, Java gives it a default constructor with no parameters that sets all fields to 0 or null
- ▶ We can also write our own constructors.

- ▶ Example:

```
public class Student {  
    String sID;  
    String name;  
    String program;  
    double gpa;  
  
    public Student(String id, String nm) {  
        sID = id;  
        name = nm;  
        program = "undeclared";  
        gpa = 0.0;  
    }  
    ...  
}
```

fields for Student class

Constructor looks similar to a method, but without a return type specified

Initializes the **sID** and **name** fields to the values of the parameters passed in

Constructor - code trace

In Student.java:

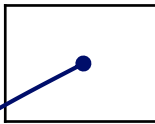
```
public Student(String id, String nm) {  
    sID = id;  
    name = nm;  
    program = "undeclared";  
    gpa = 0.0;  
}
```

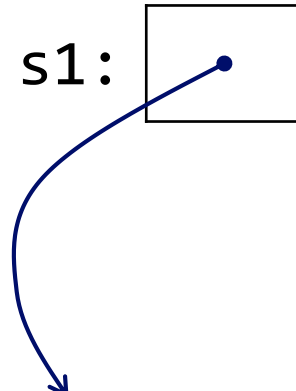
In a different program:

```
Student s1 = new Student("v00123456", "Ali Sa");
```

We are calling the constructor here:

Memory:

s1: 



sID:	"v00123456"
name:	"Ali Sa"
program:	"undeclared"
gpa:	0.0

Multiple Constructors

- ▶ A class can have multiple constructors
- ▶ Each constructor must have a unique set of parameters
 - ▶ This is similar to how methods can have the same name, but have a unique set (number and types) of parameters
- ▶ When we create a new instance of an object with the **new** keyword, and then call a constructor...
 - ▶ the constructor called will be based on the parameter list we call it with

Access Modifiers

- ▶ So far, our examples have all used the `public` keyword
- ▶ The `public` keyword is an **access modifier**
- ▶ Access modifiers:
 - ▶ **`public`**: allows accessibility by any other class
 - ▶ **`private`**: only accessible within the declared class
 - ▶ **`protected`**: only accessible by classes within the same package, or by subclasses (we will learn about these later)

Student example

- ▶ We can set the access modifiers for all of our fields to private:

```
public class Student {  
    private String sID;  
    private String name;  
    private String program;  
    private double gpa;  
  
    ...  
}
```

- ▶ Now we will need another way to work with these fields!

Accessor / Mutator methods

- ▶ Sometimes called “getters” and “setters”
- ▶ Typically, many fields within a class are not publicly accessible
- ▶ Instead, we will define methods that allow us to access (get) or mutate (set) the values of fields indirectly

```
public String getProgram() {  
    return program;  
}
```

These methods are public so can be accessed by other classes

```
public void setProgram(String newProgram) {  
    program = newProgram;  
}
```

These methods are defined within the Student class, so they can access private fields

Accessor and Mutator Example

Student.java

```
public class Student {  
    private String sID;  
    private String name;  
    private String program;  
    private double gpa;  
  
    ...  
  
    public String getName() {  
        return name;  
    }  
  
    public void setProgram(String newP) {  
        program = newP;  
    }  
}
```

StudentAnalysis.java

```
Student s1 = new Student("v00123456",  
    "Ali Sa", "Mathematics", 4.3);  
  
System.out.print(s1.sID);  
  
System.out.println(s1.getName());  
  
s1.setProgram("Biology");
```

error: sID has private access in Student

The toString method

- ▶ The toString method returns a String representation of an object
- ▶ It is up to the programmer to decide what information is present in String representation of the object
- ▶ This method is automatically called when we print out an object

toString example

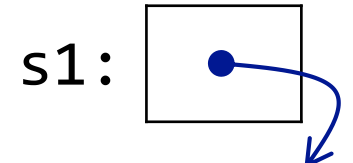
Student.java

```
public class Student {  
    private String sID;  
    private String name;  
    private String program;  
    private double gpa;  
  
    ...  
  
}
```

StudentAnalysis.java

```
Student s1 = new Student("v00123456",  
    "Ali Sa", "Mathematics", 4.3);  
  
System.out.print(s1);
```

Memory:



sID:	"v00123456"
name:	"Ali Sa"
program:	"Mathematics"
gpa:	4.3

Output: Student@5ca881b5

toString example

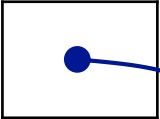
Student.java

```
public class Student {  
    private String sID;  
    private String name;  
    private String program;  
    private double gpa;  
  
    ...  
  
    public String toString() {  
        String s = name + " - " + sID;  
        return s;  
    }  
}
```

StudentAnalysis.java

```
Student s1 = new Student("v00123456",  
    "Ali Sa", "Mathematics", 4.3);  
  
System.out.print(s1);
```

Memory:

s1: 

sID:	"v00123456"
name:	"Ali Sa"
program:	"Mathematics"
gpa:	4.3

Output: Ali Sa - v00123456

The `this` keyword

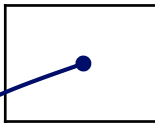
```
public class Student {  
    private String sID;  
    private String name;  
    private double gpa;  
    private String program;
```

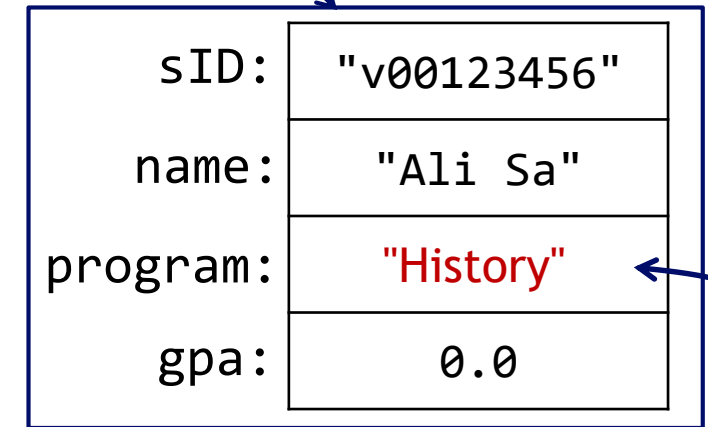
...

```
s1.setProgram("History");
```

```
public void setProgram(String newProgram) {  
    program = newProgram;  
}  
}
```

Memory:

s1: 



setProgram:

newProgram: 

The `this` keyword

```
public class Student {  
    private String sID;  
    private String name;  
    private double gpa;  
    private String program;
```

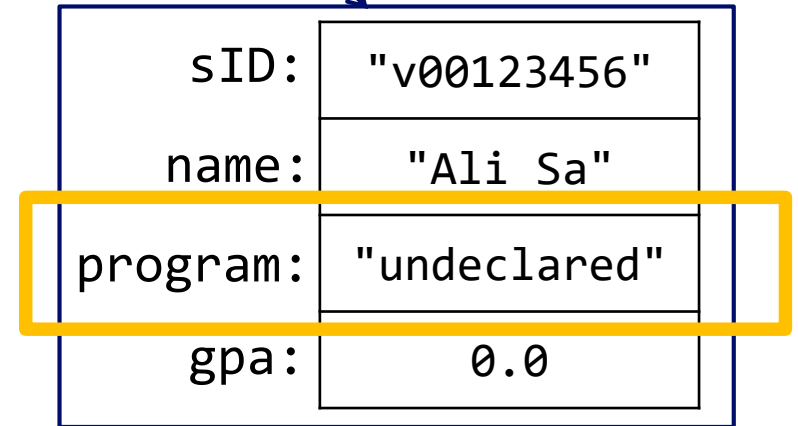
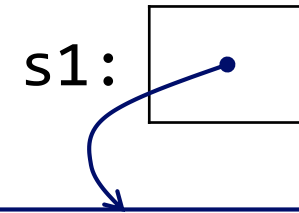
...

```
s1.setProgram("History");
```

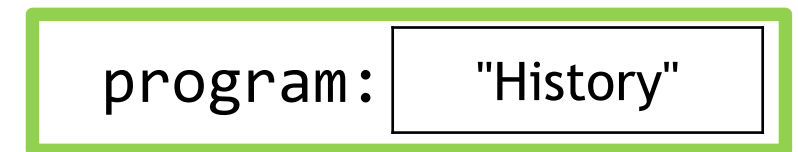
```
public void setProgram(String program) {  
    System.out.println(program);  
}  
}
```

What is output?

Memory:



setProgram:



The **this** keyword

```
public class Student {  
    private String sID;  
    private String name;  
    private double gpa;  
    private String program;
```

we use **this** to refer to something that is part of the class

...

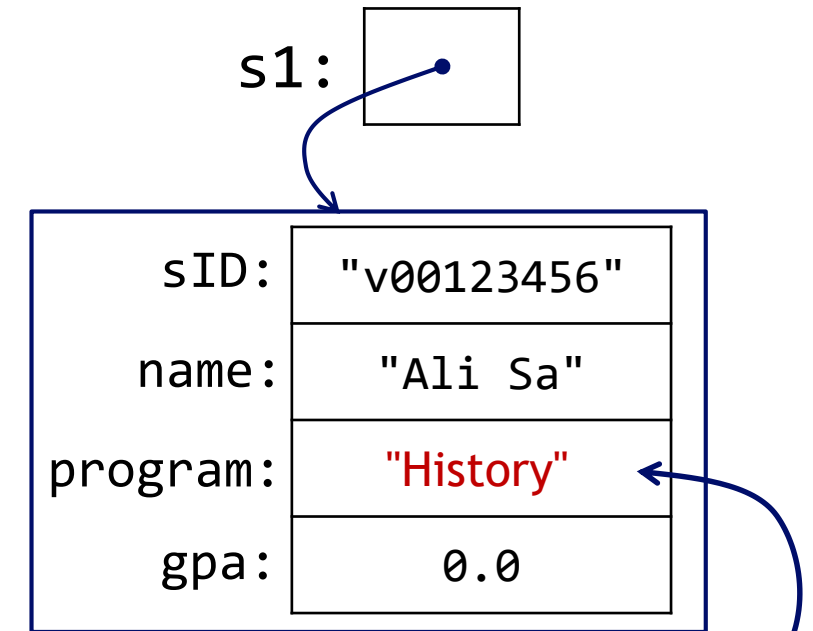
```
s1.setProgram("History");
```

```
public void setProgram(String program) {  
    this.program = program;  
}  
}
```

refers to the class instance variable (field)

refers to the parameter

Memory:



setProgram:

program: "History"

The **this** keyword

- We often use the **this** keyword in our constructors:

```
public class Student {  
    private String sID;  
    private String name;  
    private double gpa;  
    private String program;
```

We use **this** to refer to the **instance variables (fields)**. In the constructor, we want to initialize these with values passed in as **parameters**.

```
    public Student(String sID, String name, double gpa, String program) {  
        this.sID      = sID;  
        this.name      = name;  
        this.gpa       = gpa;  
        this.program   = program;  
    }
```

Static vs. Non-Static

- ▶ A **static** method means that it can be accessed without creating an object of the class
 - ▶ typical for generic methods that operate on data passed in as a parameter
 - ▶ example: the methods in our `ArrayOperations.java` file
 - ▶ **CANNOT** access instance variables within a class
- ▶ An **instance** method is defined within a class
 - ▶ instance methods are called on instance of a class
 - ▶ operate on the object's instance variables (fields)

Static vs. Non-Static

- ▶ A **static** method operates on the parameters it is passed
 - ▶ Utility methods:
 - ▶ Operations that work on the array passed in as a parameter
 - ▶ Operations you might find on a calculator
 - ▶ These aren't associated with instances of an object (or that object's fields)
- ▶ An **instance** method operates on an object and the object's fields
 - ▶ Student class:
 - ▶ update information about the student
 - ▶ Song class:
 - ▶ update information about the song's title or artist, add time to the duration, etc.

Static vs. Non-Static

- ▶ A **static** variable is not associated with each specific instance of a class
- ▶ Whereas each object has its own values for each of its **instance** variables (fields)

- ▶ Example:

```
public class BankAccount {  
    String id;  
    double savings;  
    static double interestRate = 0.05;  
}
```

Each BankAccount object will have its own id and amount of money in their savings account



There is only one interestRate that is shared by all BankAccounts

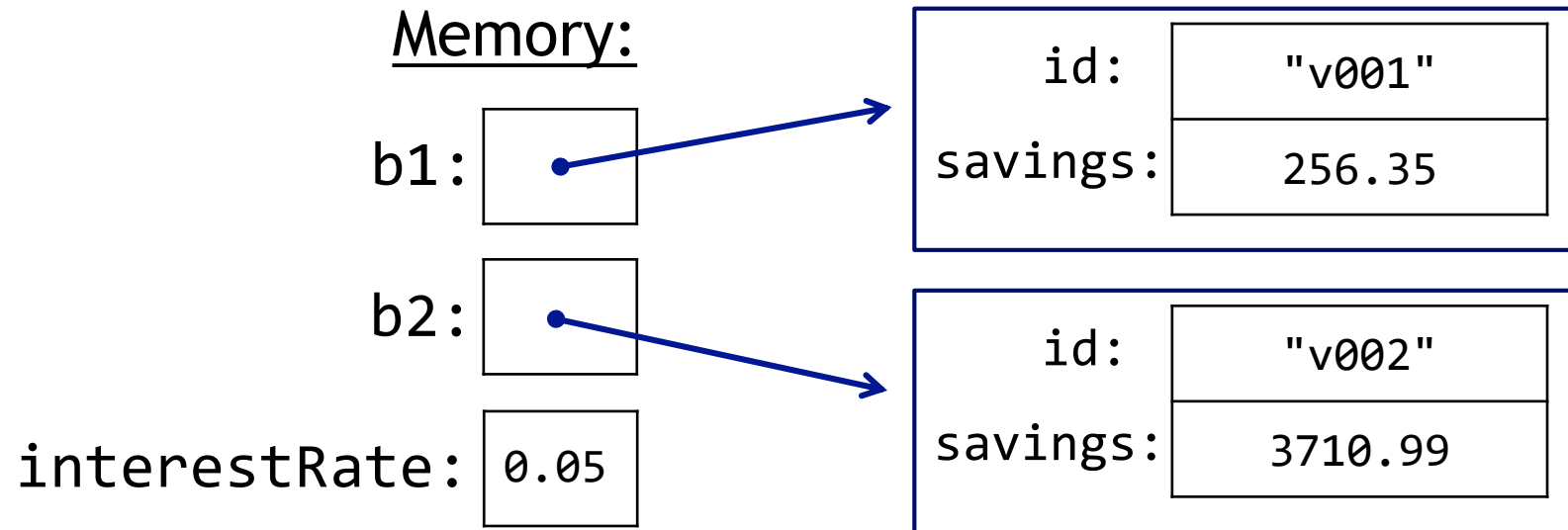


Static vs. Non-Static

► Example:

```
BankAccount b1 = new BankAccount("v001", 256.35);
```

```
BankAccount b2 = new BankAccount("v002", 3710.99);
```



Each instance of an object has memory allocated for each of its instance variables (fields)

We only need to allocate memory for static variables once

Equality example

- ▶ Often we want to check for equality in our programs
- ▶ For example:
 - ▶ Searching through a database for a particular item
 - ▶ Searching through a list to count the number of occurrences of something
- ▶ We use the `==` to determine if two primitive type variables are equal to one another
 - ▶ This works for variables of type `int`, `double`, `boolean`, etc.
 - ▶ It's not quite as simple for `Strings`, `arrays`, or `objects`

Equality example

► It all comes down to how the data is stored in memory

► For primitive types:

```
int num1 = 8;
```

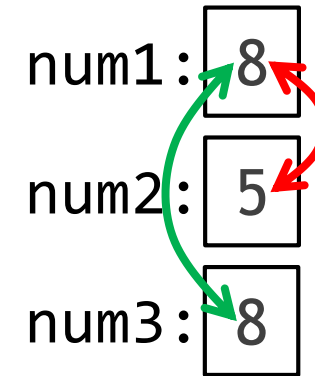
```
int num2 = 5;
```

```
int num3 = 8;
```

► num1 == num2 evaluates to false

► num1 == num3 evaluates to true

Memory:



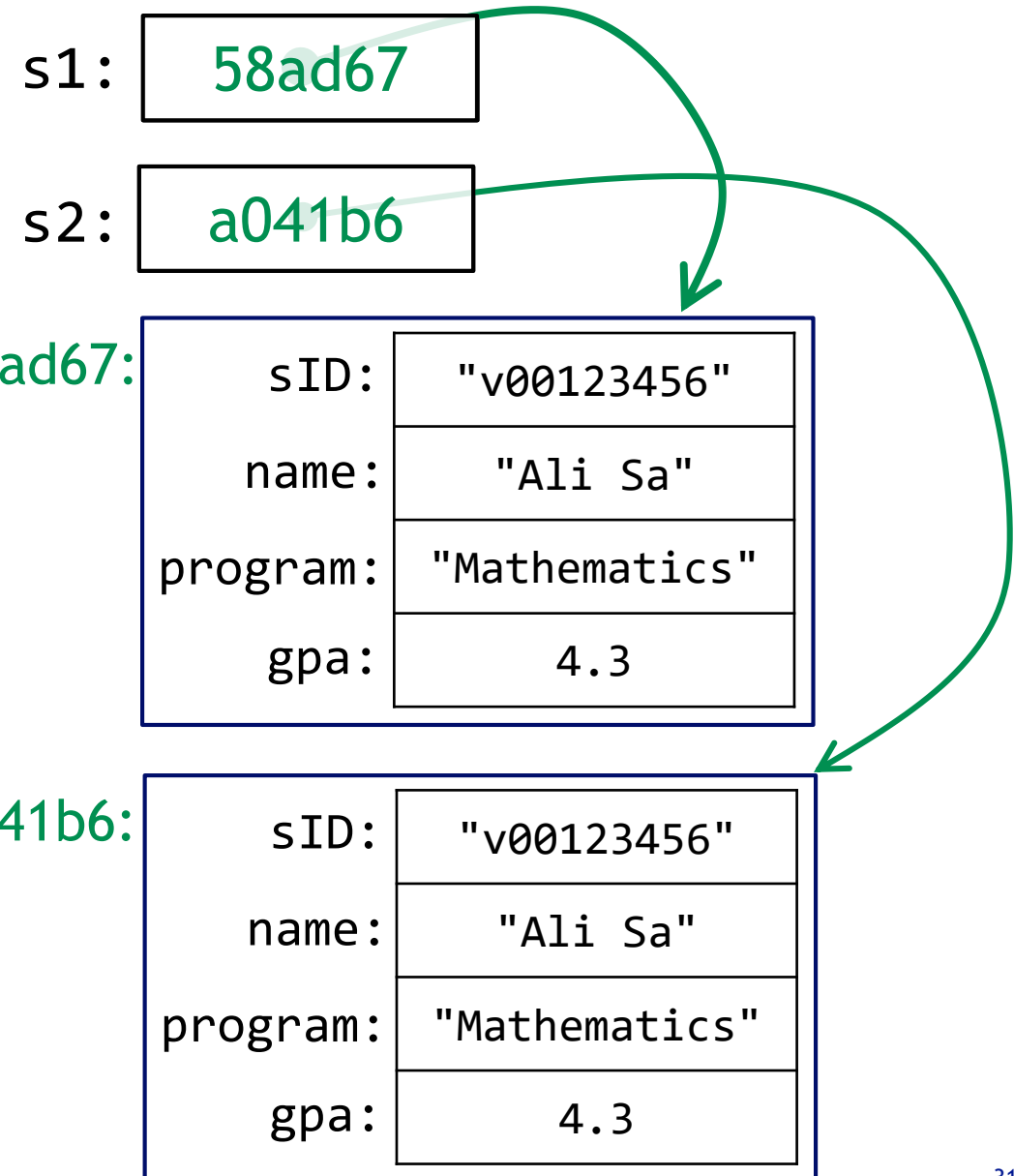
Equality example

- ▶ It all comes down to how the data is stored in memory
- ▶ For objects, Strings, and arrays, the same thing happens
 - ▶ Using `==` still compares two values in memory...
- ▶ But we need to consider what the values of our variables are
 - ▶ Remember that for these types the variable is referencing a location in memory where the data we associate with the particular object is stored
 - ▶ So although it is possible to use `==` to compare two objects, the operation likely isn't doing what we intended it to do
 - ▶ Let's take a look

Equality example

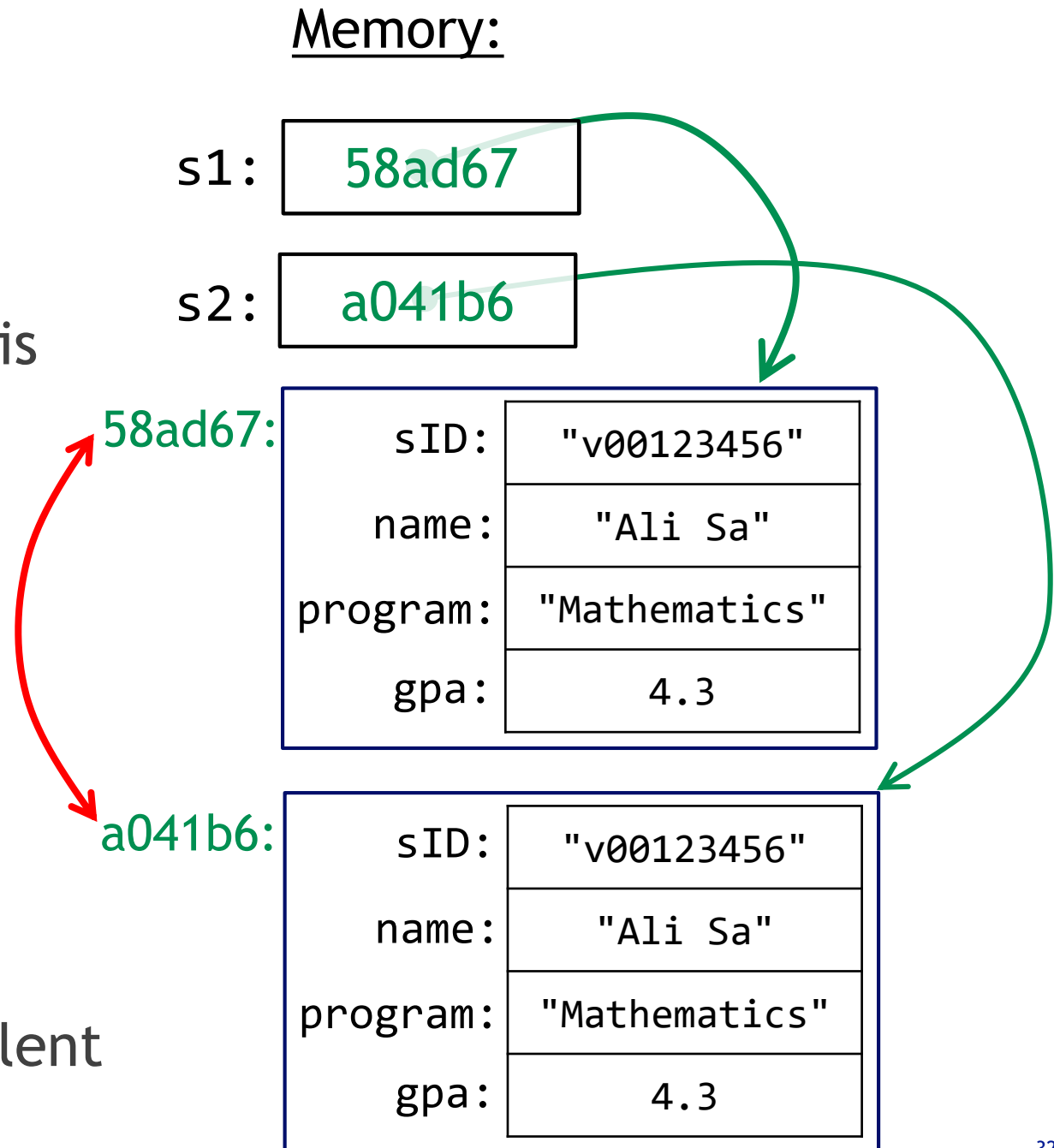
- ▶ Assume we have two student objects
 - ▶ And associate both s1 and s2 with Ali
- ▶ We can see they are equal
 - ▶ but == determines if the two values stored in memory are equivalent
- ▶ What are the values of s1 and s2?
 - ▶ s1 and s2 store the memory address of where the object data is being stored
 - ▶ We typically visualize this as an arrow pointing to the location in memory

Memory:



Equality example

- ▶ So let's consider what `==` does here:
- ▶ `s1 == s2` is checking if the value `58ad67`, which is a memory address, is equal to the value `a041b6`
 - ▶ The two addresses are not the same!
 - ▶ So `s1 == s2` evaluates to false
 - ▶ Which likely is not what we wanted or expected
- ▶ Instead we will add a method to the Student class that allows us to determine if two students are equivalent

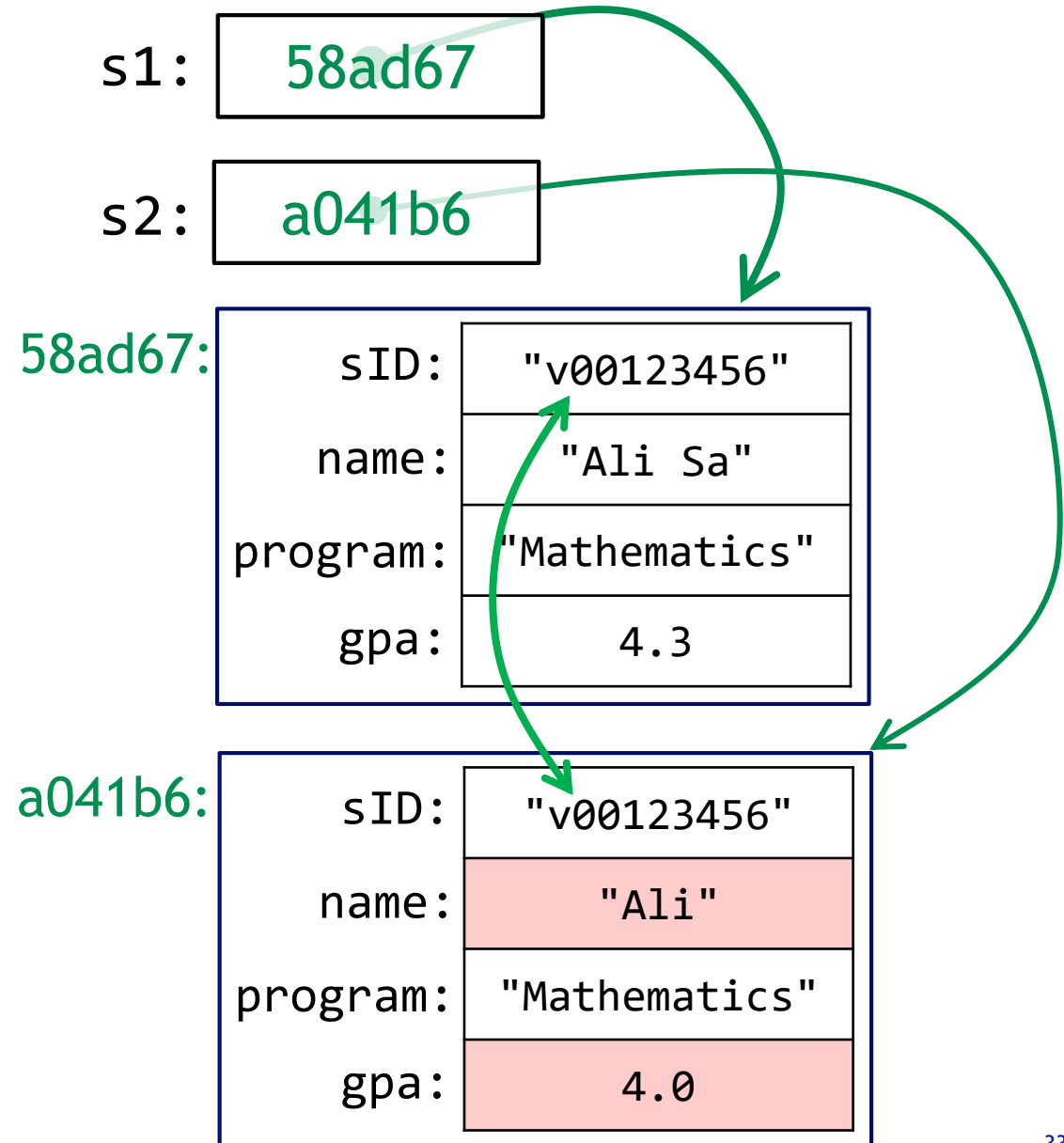


Equality example

Student.java

```
public class Student {  
    private String sID;  
    private String name;  
    private String program;  
    private double gpa;  
  
    ...  
  
    public String getSID() {  
        return sid;  
    }  
  
    public boolean equals(Student other) {  
        return sID.equals(other.getSID());  
    }  
}
```

Memory:



Equality summary

- ▶ The String class also has an `equals` method

- ▶ Key takeaways:

- ▶ We want to call the `equals` method when comparing two objects, not `==`

- ▶ For example:

```
Student s1 = new Student("v00123456", "Ali Sa", "Mathematics", 4.3);
```

```
Student s2 = new Student("v00123456", "Ali Sa", "Mathematics", 4.3);
```

`s1 == s2` evaluates to false

`==` compares if `s1` and `s2` are referencing the same location in memory, probably NOT what we want

`s1.equals(s2)` evaluates to true

`equals` calls the object's `equals` method, where the programmer specifies the equivalence relation between two objects