Software organization

- Well-designed software is organized based on:
  - Modularity: Components are independent modules.
  - Reusability: Components are reusable.
  - Integration: Components can be easily integrated with one another.
  - Abstraction: Irrelevant details, and details about how a component works, are hidden from the clients of the component.

- Object-Oriented Programming (OOP) is a set of software development techniques from analysis to implementation, to organize software based these principles.

- In OOP, modules are implemented as *classes*.

Objects and classes

- A class is a "type" of objects. Objects are the values of a class.
- A class is defined by the attributes shared by all its objects.
- In analysis we should:
  - Discover the classes of objects involved (physical or abstract,) and
  - Identify the attributes of those classes.
- These translate into code as "class definitions"

```
public class ClassName
{
    Attribute definitions

    Method definitions
}
```

Class definition structure

- Attribute definitions

  type variable;

where *type* is either a primitive data type (`int`, `boolean`, etc.) or the name of a user-defined class.

Class definition structure (contd.)

- Method definitions

```
type method_name(list_of_parameters)
{
    statements;
}
```

where *type* is either `void` (the method doesn't return anything,) a primitive data type or a user-defined data type. The *list_of_parameters* is of the form

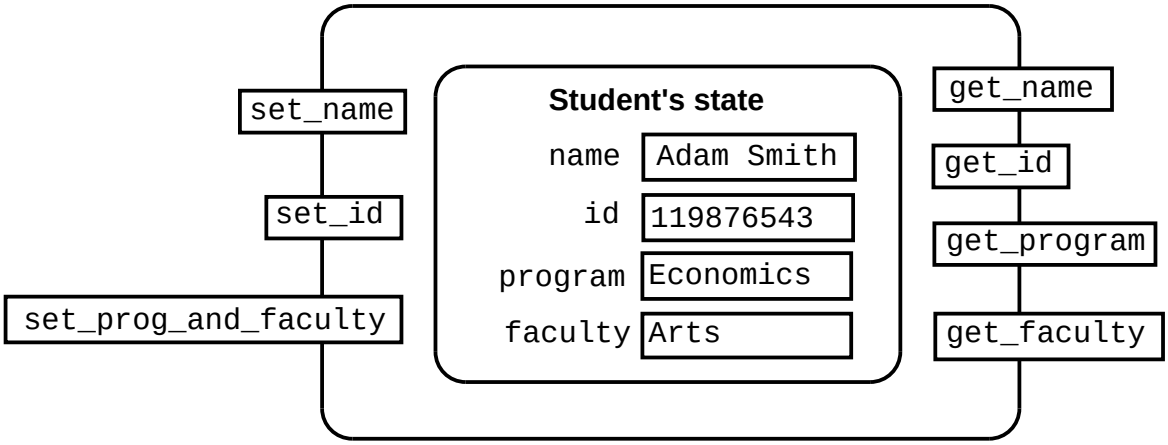
```
type1 arg1, type2 arg2, ..., typen argn
```

Example

```
public class Student
{
    String name;
    long id;
    String program;
    String faculty;

    void set_name(String s)
    {
        name = s;
    }

    void set_id(long num)
    {
        id = num;
    }

    // Continues below ...
```

```
    String get_name()
    {
        return name;
    }

    long get_id()
    {
        return id;
    }

    void set_prog_and_faculty(String p,
                              String f)
    {
        program = p;
        faculty = f;
    }

    String get_program()
    { return program; }

    String get_faculty()
    { return faculty; }
} // Class Student ends here.
```

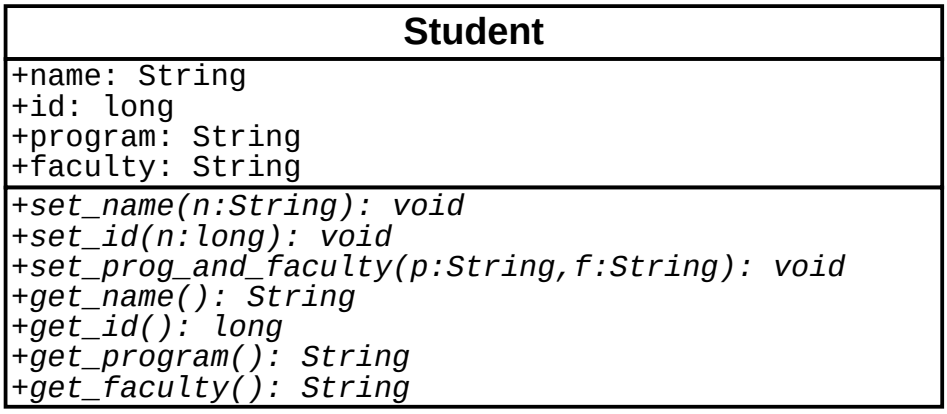
An object of the Student class

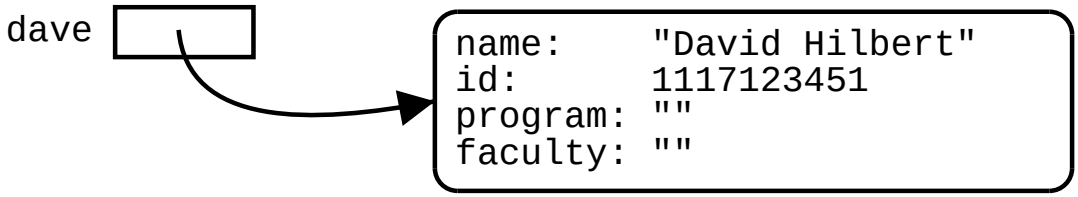


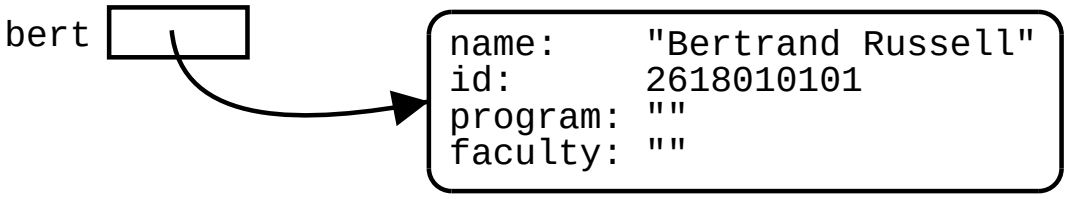
Objects are not classes

- A class can be thought of as a data type. Its values are objects.
- An *object* is an *instance* of a class.
- An object has its own separate identity and its own separate state.
- Each object is stored in different memory locations.

Individual identity of objects







Creating objects

- To create objects of a given class:

  First: Declare a variable of that type:

  ```
  class_name variable;
  ```

  Second: Assign the variable a new instance, using the new keyword:

  ```
  variable = new class_name();
  ```

- Example

  ```
  Student dave;

  dave = new Student();
  ```

- The two can be done in one line:

  ```
  Student bert = new Student();
  ```

Accessing attributes

- The attributes of an object can be accessed directly using the dot operator:

  variable.attribute

...but only if the attribute exists in the class of the variable.

- Example:

```
dave.name = "David Hilbert";
dave.id = 111712345l;
System.out.println(dave.name);
System.out.println(dave.id);

bert.name = "Bertrand Russell";
bert.id = 261801010l;
System.out.println(bert.name);
System.out.println(bert.id);
```

Sending messages to objects

- To interact with an object we send it a message by *calling*, or *invoking* one its methods.

- Calling a method is done by using the dot operator, and passing parameters or arguments (if any):

  variable.*method_name*(*arguments*)

where the type of *variable* is a class which has a method called *method_name*, and *arguments* is a coma-separated list of values whose type matches those of the method's parameters.

Sending messages (contd.)

- For example:

```
bert.set_prog_and_faculty(“Philosophy”, “Arts”);

dave.set_id(009876543);
```

- A method call

```
a.m(b, c, d);
```

could be interpreted as “sending the message `m` to the object `a` with arguments `b`, `c`, and `d`.”

Method calls in context

- There are two forms of method calls:
  - Method call as a statement
  - Method call as an expression
- A method call is a statement if its return type is `void`, otherwise it is an expression.
- If a method call is an expression, it must appear in a context that allows expressions, such as:

  A. the right hand-side of an assignment:

  ```
  long n = dave.get_id();
  String s = dave.get_program();
  ```

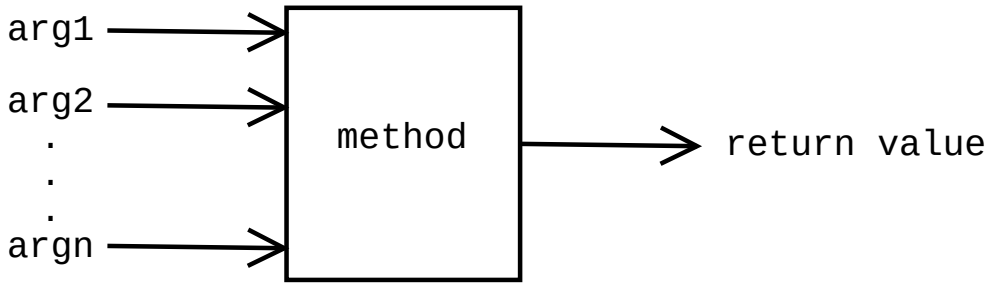
  B. ...or, the argument of another method:

  ```
  System.out.println(dave.get_id());
  bert.set_id(dave.get_id());
  ```

- But the types **must** match!

Methods as functions

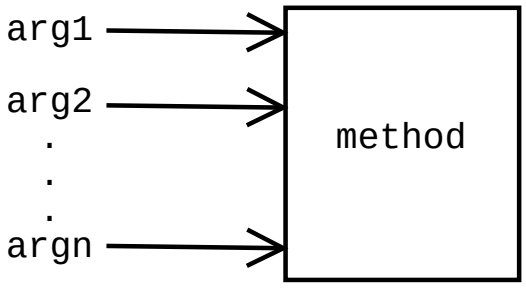
- Methods can be viewed as a "black box" with inputs and outputs:



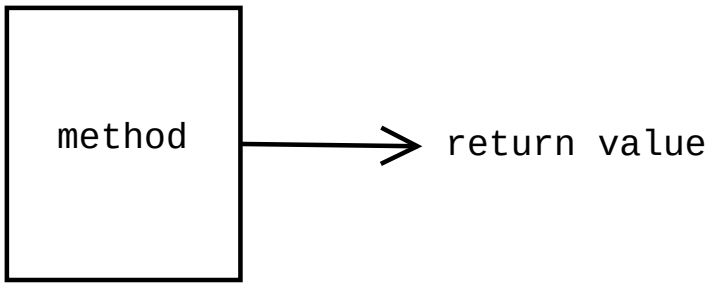
- There are three kinds of methods:
  - Mutators: Modify the state of objects,
  - Accessors: Return information about the object,
  - Constructors: Initialize a newly created object.

Method types

- Mutators are usually `void` methods, which do not return anything, but modify the state of the object:



- Accessor methods may only return values without expecting any arguments as input:



Constructors

- Special methods, whose syntax is given by

```
class_name(list_of_arguments)
{
    statements;
}
```

- For example:

```
public class Student {
    //...
    Student(String n, long i)
    {
        name = n;
        id = i;
    }
    //...
}
```

Constructors (contd.)

- A constructor method gets executed when a new object of the class gets created using the new keyword. Therefore, the general syntax for the expression used to create objects is:

```
new class_name(list_of_actual_arguments);
```

- For example

```
Student al;
al = new Student(“Alan Turing”, 110011223331);
```

Software clients

- A *client* of a class C (or software component in general) is any other class (or software component) which uses C.

- For example:

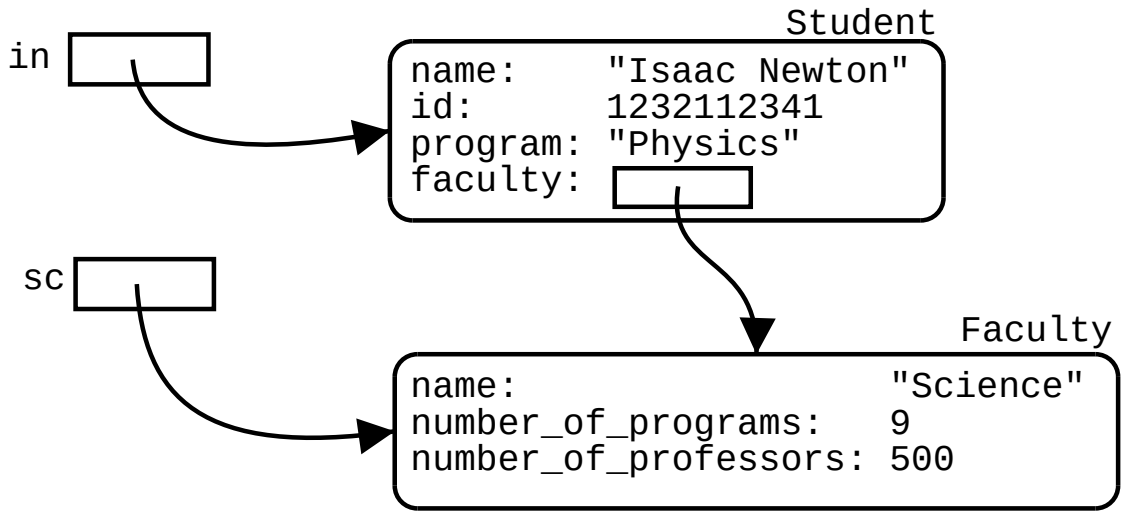
```
public class StudentDatabase {
  public static void main(String[] args)
  {
    Student dave = new Student(“David Hilbert”,
                               1223334444);
    Student bert = new Student(“Bertrand Russell”,
                               1111222334);
    dave.set_prog_and_faculty(“Math”,“Science”);
    //...
  }
}
```

Classes are data types

```
public class Faculty
{
    String name;
    int number_of_programs, number_of_professors;
    //...
}

public class Student
{
    String name;
    long id;
    String program;
    Faculty faculty;
    //...
    void set_prog_and_faculty(String p, Faculty f)
    {
        program = p;
        faculty = f;
    }
    //...
}
```

```
public class StudentDatabase
{
  public static void main(String[] args)
  {
    Faculty sc = new Faculty();

    sc.name = ``Science'';

    Student in = new Student(``Isaac Newton'',
                             1232112341);

    in.set_prog_and_faculty(``Physics'', sc);

    //...
    System.out.println(sc.name);
    System.out.println(in.name);
  }
}
```

Object structure in memory




```
in.set_prog_and_faculty("Physics", new Faculty());
```

doesn't create the variable `sc`, but then, the `Faculty` object cannot be shared between different `Student` objects.

Scope

- Different classes can have attributes and methods which have the same names.

- For example given the following class definition

```
public class C {
    int a;
    //...
}
```

the variables `x.a` and `y.a` are different memory locations in the following client:

```
public class D {
    void m()
    {
        C x = new C();
        C y = new C();
        x.a = 3;
        y.a = 5;
    }
}
```

Scope (contd.)

- This also applies if the attributes are in different classes:

```
public class C {
  int a;
  // ...
}
public class E {
  int a;
  // ...
}
public class D {
  void m()
  {
    C x = new C();
    E y = new E();
    x.a = 3;
    y.a = 5;
  }
}
```

Scope (contd.)

- The *scope* of a variable, a parameter, an attribute or a method is the part of the program that can access that variable, attribute or method.
- The scope of a parameter of a method is the body of the method.
- Variables declared in the body of a method are called *local variables*, which means that their scope is only the body of the method.
- The direct scope of an attribute of a class or a method is the class itself (e.g. the direct scope of `id` is the `Student` class.)
- However, the indirect scope of an attribute of a class or a method is the rest of the program (e.g. the `id` attribute can be accessed by other clients with the expression `var.id`, where var is of type `Student`.)