Ch 7 Designing Java Classes

We have used a number of Java classes: Scanner, String, Random, Math, Character ….

Now we consider defining our own classes

A couple of quick examples:

- PlayingCard
- Word

Example a PlayingCard class

```
public class PlayingCard {
    private String suit;
    private String face;
```

Fields/data describing a card

```
    public PlayingCard(String s, String f){
        suit = s;
        face = f;
    }
```

Constructing/initializing a card

```
     public String toString(){
          return face+” of “+ suit;
      }
}
```

How a card is displayed

```
public class UsePlayingCards {
    public static void main(String[] args) {

        PlayingCard p1 = new PlayingCard(…);
        System.out.println(p1); }
}
```

Example a Word class

```
public class Word {
    private String text;
    private int frequency;
```

Fields/data describing a word

```
    public Word(String w){
        text = w;
        frequency = 1;
    }
```

Constructing/initializing a word

```
     public String toString(){
         return text;
     }
```

How a word is displayed

```
}
```

```
public class ProcessWords {
    public static void main(String[] args) {

        Word w = new Word("Java");
        System.out.println(w); }
}
```

Class structure

Classes comprise fields and methods

Fields:

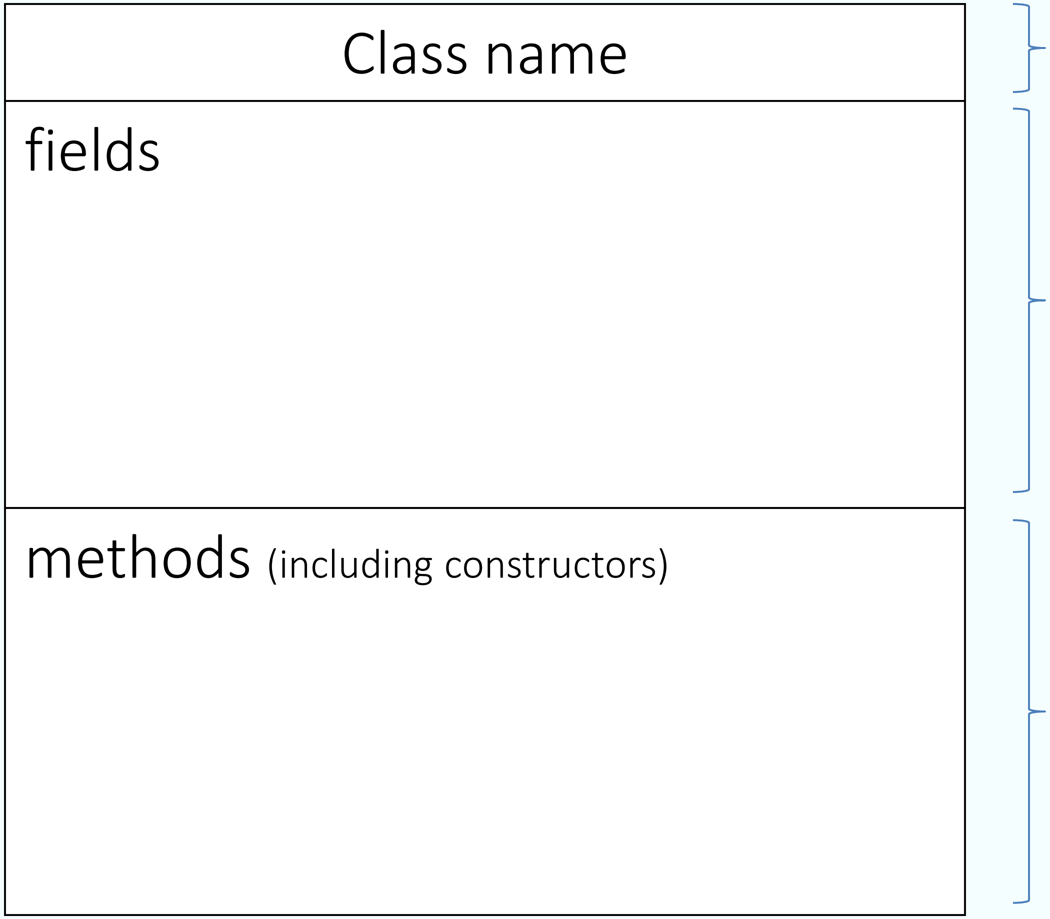
Things that describe the class
  or describe instances (i.e. objects)
e.g. student number, first name, last name, gender, …

Methods:

constructors, getters, setters, other…
e.g. getFirstName(), setFirstName(), equals()

a getter/accessor a setter/mutator

UML Diagram of a Class



Class name

fields

methods (including constructors)

A class has a name,

Fields
(called variables in
Java code)

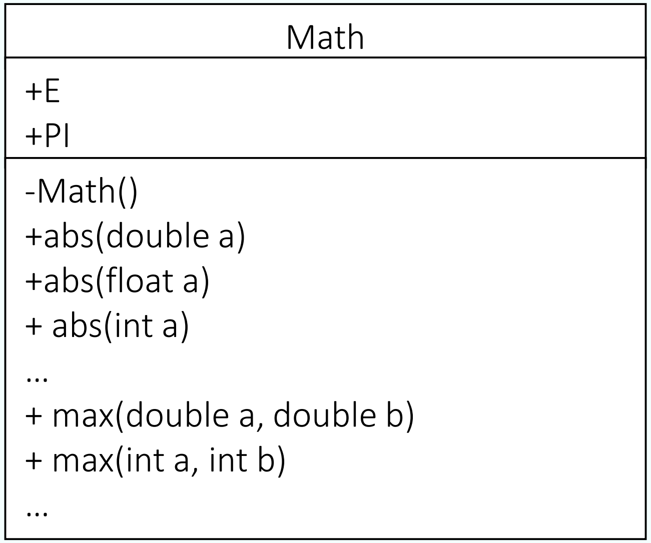
Methods
(A method may have
local fields)

e.g. Math & Random classes

A quick look at two classes we have used: Math and Random

Math provides some useful utility methods.
We use it without instantiating an object.

```
double area = Math.PI * Math.pow(r,2);
```



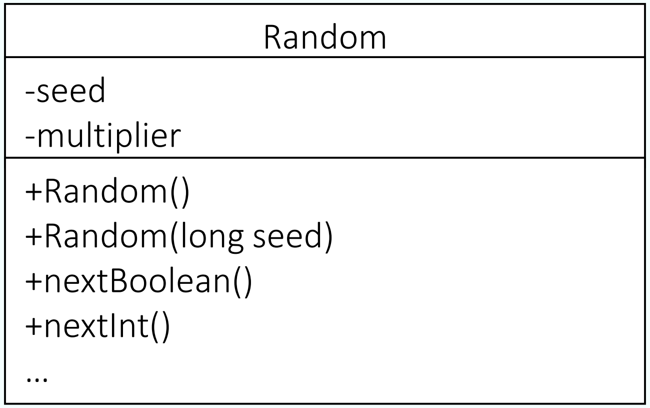
Math

+E
+PI

-Math()
+abs(double a)
+abs(float a)
+ abs(int a)
…
+ max(double a, double b)
+ max(int a, int b)
…

Random lets us use random sequences.
To utilize this we must instantiate objects.

```
Random die = new Random();
int toss = die.next(6)+1;
```



Random

-seed
-multiplier

+Random()
+Random(long seed)
+nextBoolean()
+nextInt()
…

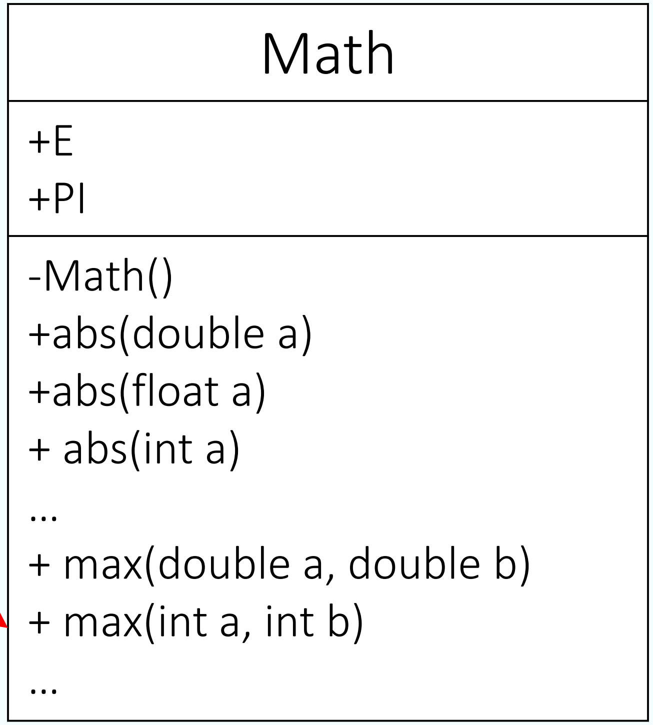
e.g. Math class

Math has two **static** fields

Math has a ***private*** constructor

You cannot instantiate a Math object

Math has many **static** methods



Math

+E
+PI

-Math()
+abs(double a)
+abs(float a)
+ abs(int a)
…
+ max(double a, double b)
+ max(int a, int b)
…

To use π you write

```
Math.PI
```

To use the static method `max` you write

```
Math.max(n1, n2)
```

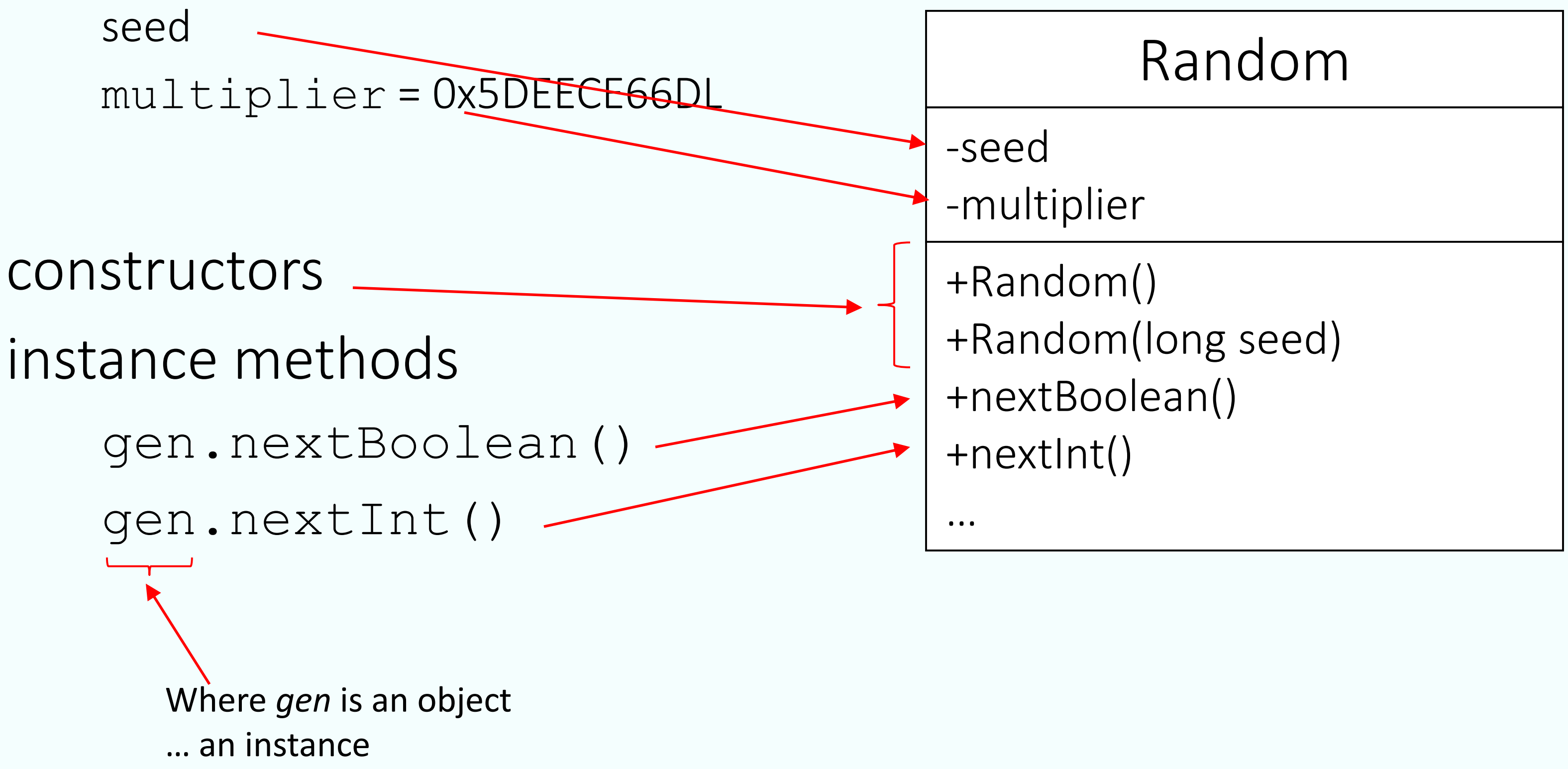
Here we specify the name of the class

e.g. Random class

We must instantiate an object to get a random sequence

```
    Random gen = new Random ();
```

Random has *some private* instance fields



seed

`multiplier` = 0x5DEECE66DL

constructors

instance methods

```
    gen.nextBoolean()
    gen.nextInt()
```

Where *gen* is an object
… an instance

Random

-seed
-multiplier

+Random()
+Random(long seed)
+nextBoolean()
+nextInt()
…

Fields

Fields may be primitive variables

Or, they may be of some other type

e.g. String, PlayingCard, Word

May be public or private

- public – anyone can use it
- private – limited access

Methods

- Methods are either:
  - value-returning

    must have a return statement

    e.g. getters ← naming convention is …

  - void

    no return statement

    e.g. setters ← naming convention is …

Methods

- public vs private

  public - anyone can use it

  private - special cases

    Math constructor is private – you cannot instantiate a Math object … try to do it

Methods

- All classes should have

    equals(…)

    toString()

equals Method

equals(…)

- Value-returning
- Returns a boolean
- Usually an equals method is designed for a class. Designer must determine the condition when two objects are considered equal.
- E.g. String class has an equals method

```
 string1.equals(string2)
“abc”.equals(“xyz”)  returns false
“abc”.equals(“abc”)  returns true
```

toString Method

toString()

- Value-returning
- Returns a string
- A method automatically called when an object is displayed

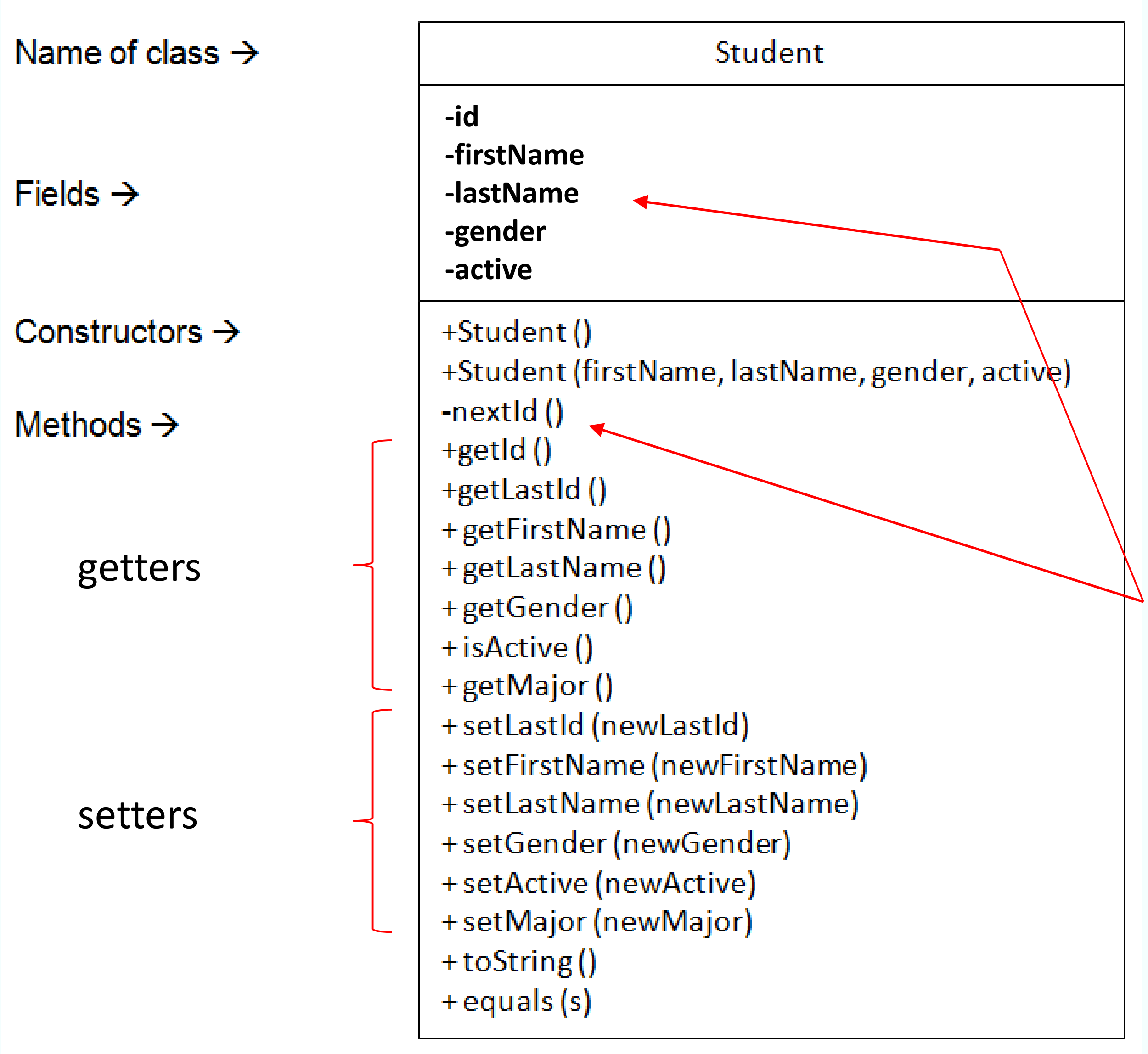
  `E.g. System.out.println(myObject);`
- The designer of a class determines what it returns
- E.g. ArrayList has a toString() method … result is of the form:

  [$\texttt{object}_1$, $\texttt{object}_2$, … $\texttt{object}_n$]

Text Example - Student

Consider the student class in the text →

Class Diagram for Student



Name of class →

Student

Fields →

-id
-firstName
-lastName
-gender
-active

Constructors →

+Student ()
+Student (firstName, lastName, gender, active)

Methods →

-nextId ()

getters

+getId ()
+getLastId ()
+ getFirstName ()
+ getLastName ()
+ getGender ()
+ isActive ()
+ getMajor ()

setters

+ setLastId (newLastId)
+ setFirstName (newFirstName)
+ setLastName (newLastName)
+ setGender (newGender)
+ setActive (newActive)
+ setMajor (newMajor)

+ toString ()
+ equals (s)

show 1, 2, or 3 compartments/ info as needed

+ means there is public access to the method

- means there is no public access to the field or method

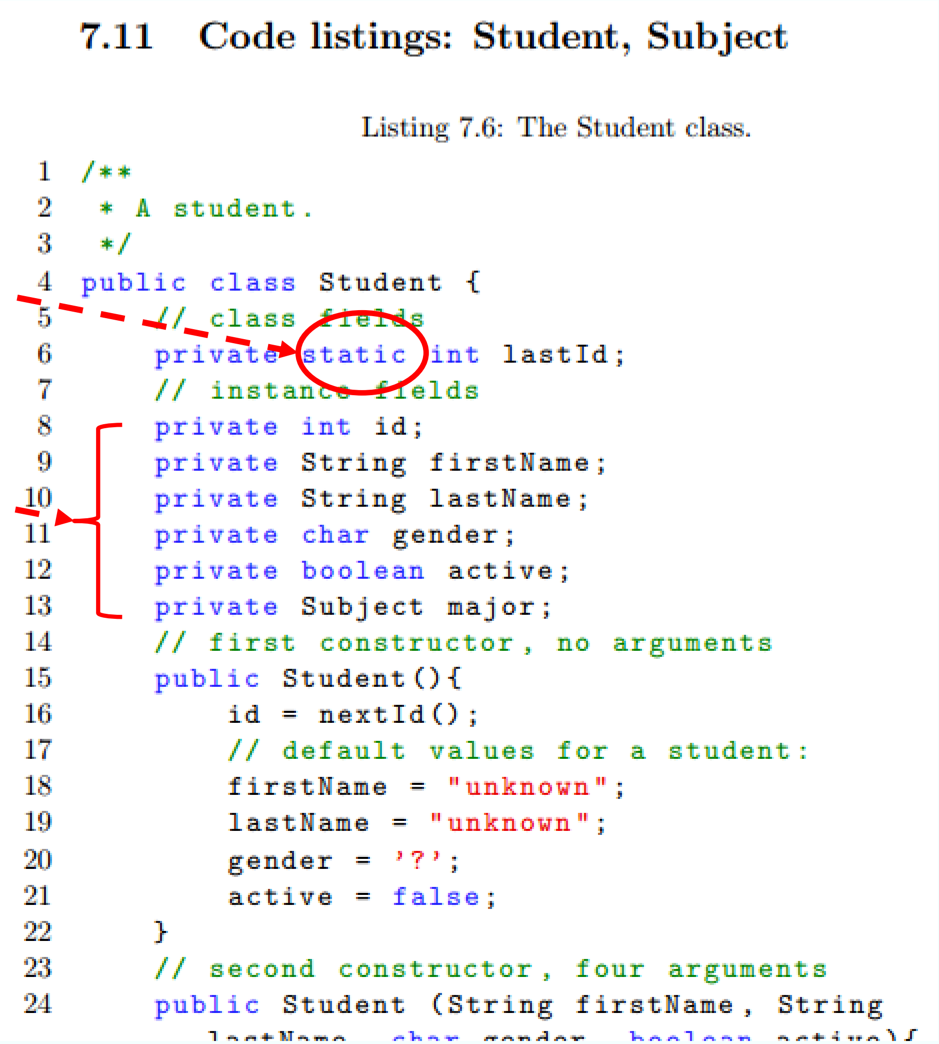
Java code for Student - fields

instance vs class

e.g. consider Student class

Which fields are class?

Which fields are instance?



7.11 Code listings: Student, Subject

Listing 7.6: The Student class.

```
/**
 * A student.
 */
public class Student {
    // class fields
    private static int lastId;
    // instance fields
    private int id;
    private String firstName;
    private String lastName;
    private char gender;
    private boolean active;
    private Subject major;
    // first constructor, no arguments
    public Student(){
        id = nextId();
        // default values for a student:
        firstName = "unknown";
        lastName = "unknown";
        gender = '?';
        active = false;
    }
    // second constructor, four arguments
    public Student (String firstName, String
        lastName, char gender, boolean active){
```

Java code for Student - fields

instance vs class

Instance ≡ an *object*

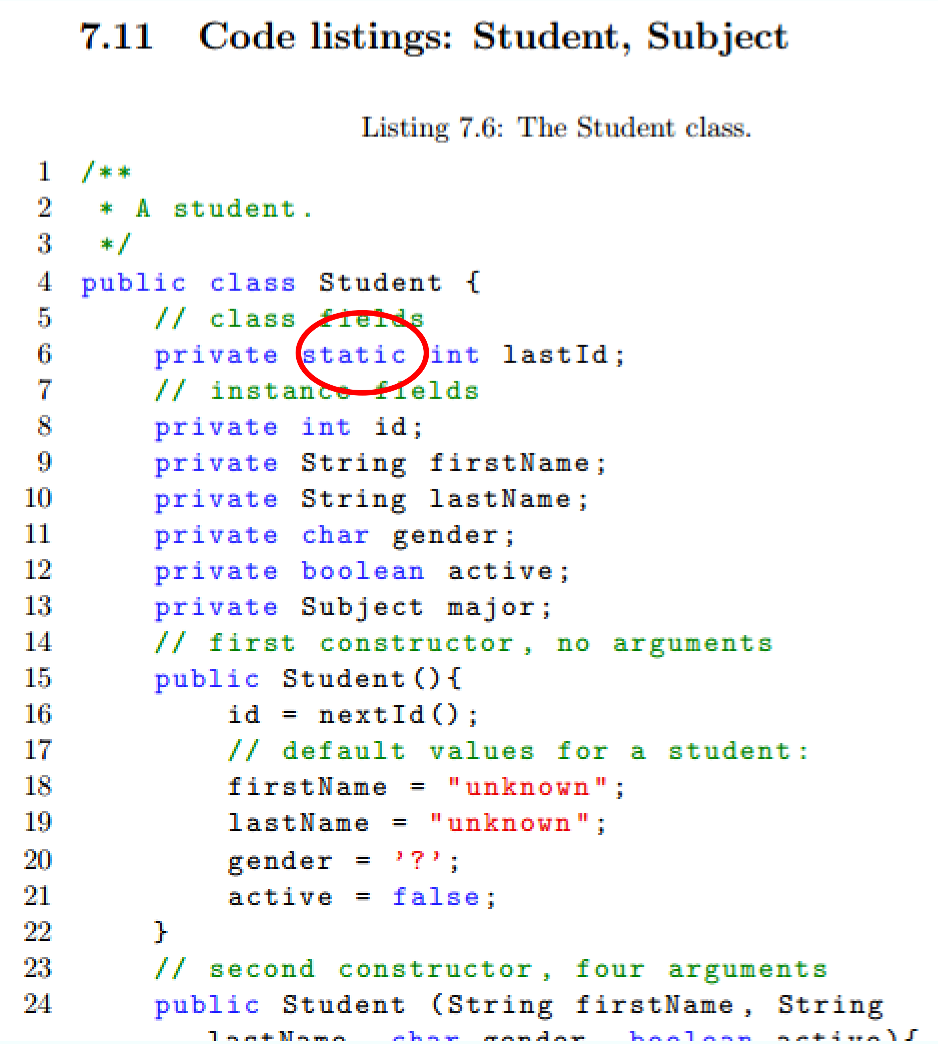
Static field ≡ *class-level* field

Regardless of the number of students there is only one `lastId` field.

It is a class-level field that is shared

by <u>all</u> Student instances

There are `id`, `firstName`, `lastName`, `gender`, `active`, and `major` fields for <u>each</u> Student *instance* .
So each student can have different values.

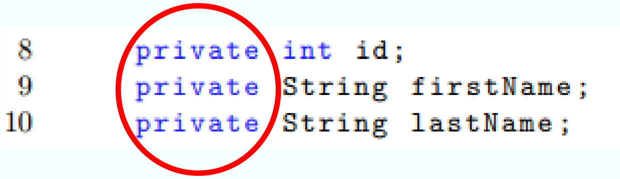


7.11 Code listings: Student, Subject

Listing 7.6: The Student class.

```
/**
 * A student.
 */
public class Student {
    // class fields
    private static int lastId;
    // instance fields
    private int id;
    private String firstName;
    private String lastName;
    private char gender;
    private boolean active;
    private Subject major;
    // first constructor, no arguments
    public Student(){
        id = nextId();
        // default values for a student:
        firstName = "unknown";
        lastName = "unknown";
        gender = '?';
        active = false;
    }
    // second constructor, four arguments
    public Student (String firstName, String
        lastName, char gender, boolean active){
```

Java code for Student - fields



```
    private int id;
    private String firstName;
    private String lastName;
```

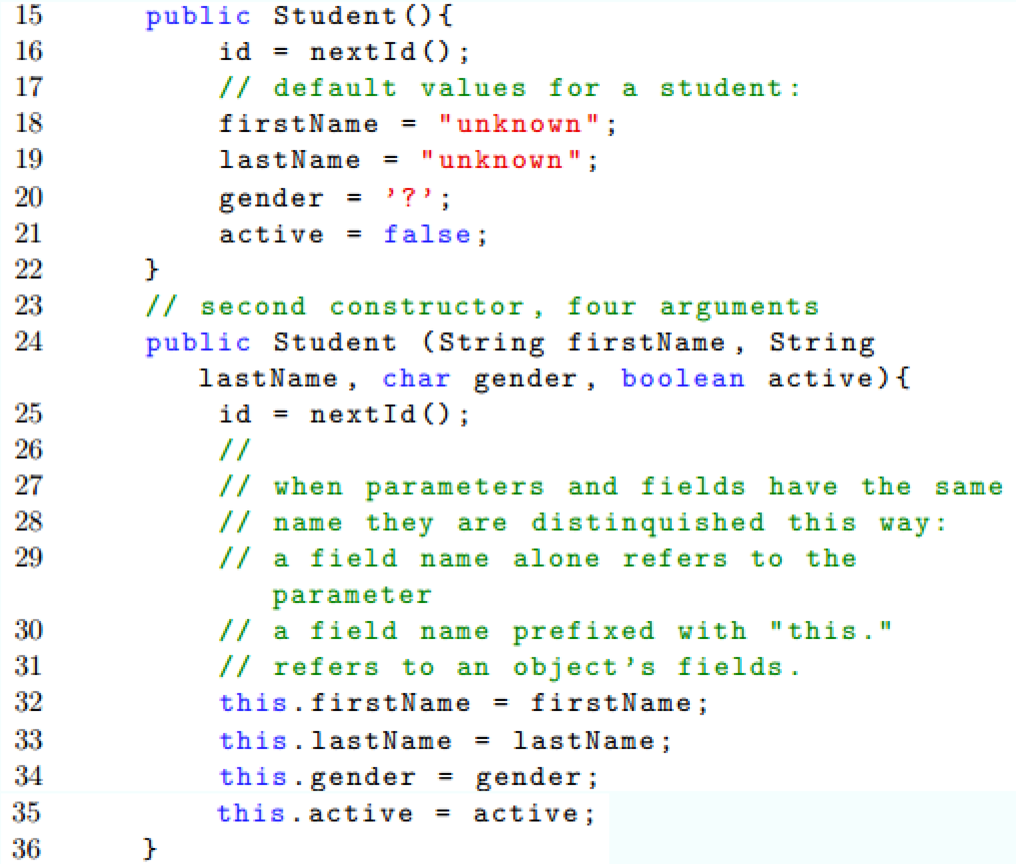
private vs public

- private:
    - only directly accessible from within the class/object,
    - and from outside the class via getters/setters
- public : accessible from anywhere

A design principle is to make **fields private**

but give **public access to the getters** and setters (a later slide)

Java code for Student - constructors



```java
    public Student(){
        id = nextId();
        // default values for a student:
        firstName = "unknown";
        lastName = "unknown";
        gender = '?';
        active = false;
    }
    // second constructor, four arguments
    public Student (String firstName, String
       lastName, char gender, boolean active){
        id = nextId();
        //
        // when parameters and fields have the same
        // name they are distinquished this way:
        // a field name alone refers to the
           parameter
        // a field name prefixed with "this."
        // refers to an object's fields.
        this.firstName = firstName;
        this.lastName = lastName;
        this.gender = gender;
        this.active = active;
    }
```

The *no-arg* constructor

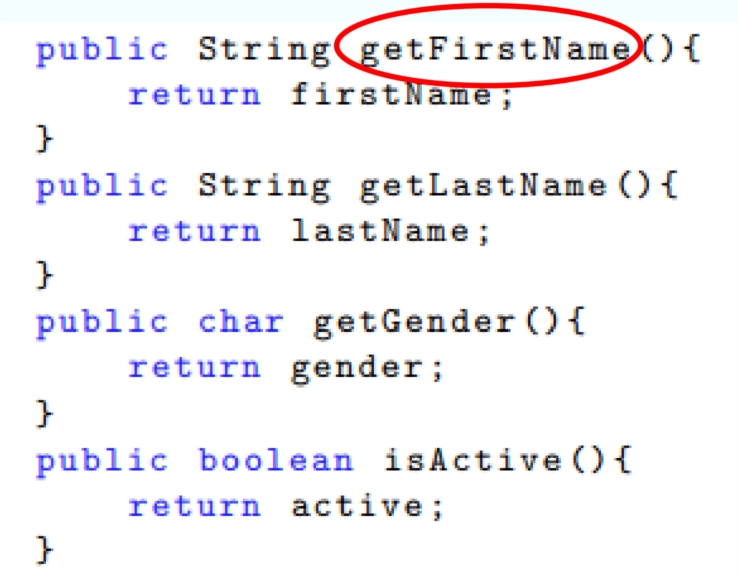
Constructor with 4 parameters
-a *4-arg* constructor

Use as many constructors as your application requires.
Constructors differ in the number and type of parameters.

Java code for Student - getters

Notice

Getters (also called *accessors*) for most private fields



```
public String getFirstName(){
    return firstName;
}
public String getLastName(){
    return lastName;
}
public char getGender(){
    return gender;
}
public boolean isActive(){
    return active;
}
```

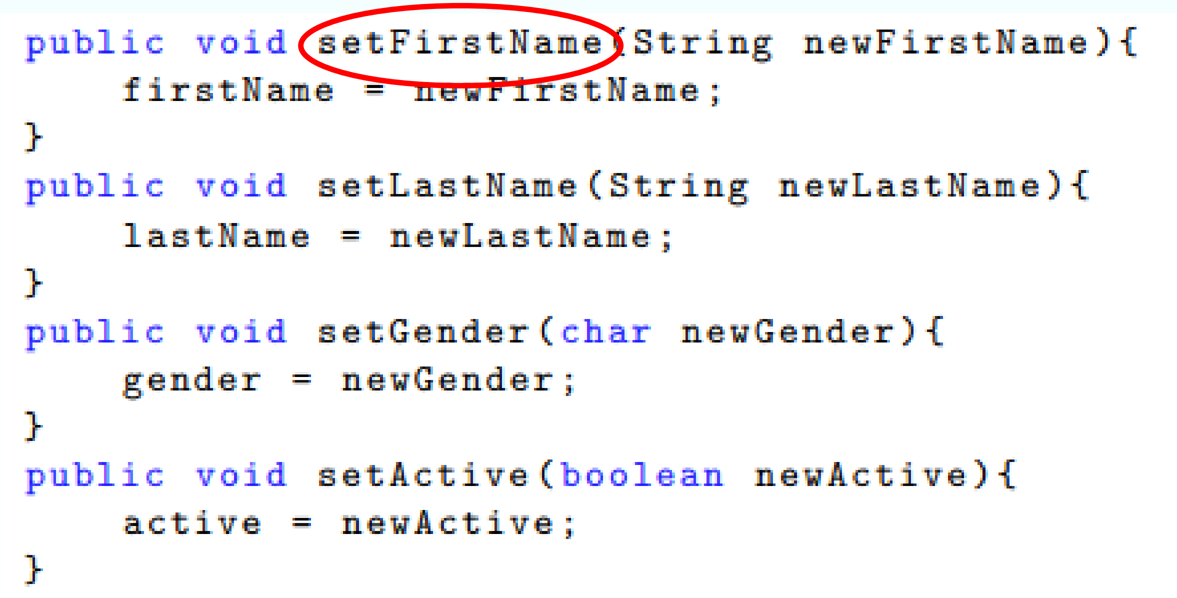
Naming convention:
Start with “get” followed by the field name but this starts with a capital letter

Naming convention for *boolean*:
Start with “is” followed by the field name but this starts with a capital letter

Java code for Student - setters

Notice

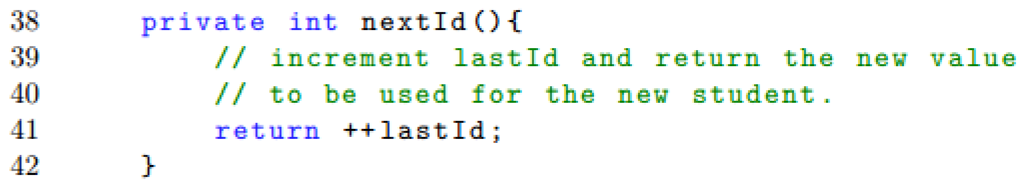
Setters (also called *mutators*) for most private fields



```java
public void setFirstName(String newFirstName){
    firstName = newFirstName;
}
public void setLastName(String newLastName){
    lastName = newLastName;
}
public void setGender(char newGender){
    gender = newGender;
}
public void setActive(boolean newActive){
    active = newActive;
}
```

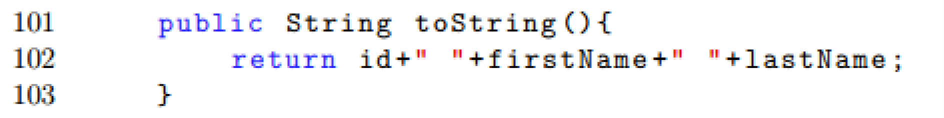
Naming convention:
Start with “set” followed by the field name but this starts with a capital letter

Java code for Student – other methods



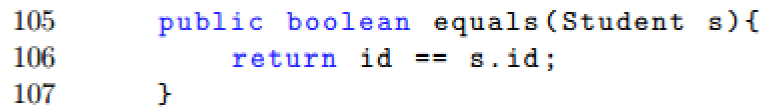
```java
    private int nextId(){
        // increment lastId and return the new value
        // to be used for the new student.
        return ++lastId;
    }
```

private method nextId
Used to control the id assigned
to a new student object



```java
    public String toString(){
        return id+" "+firstName+" "+lastName;
    }
```

toString
Executes when a student is printed



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

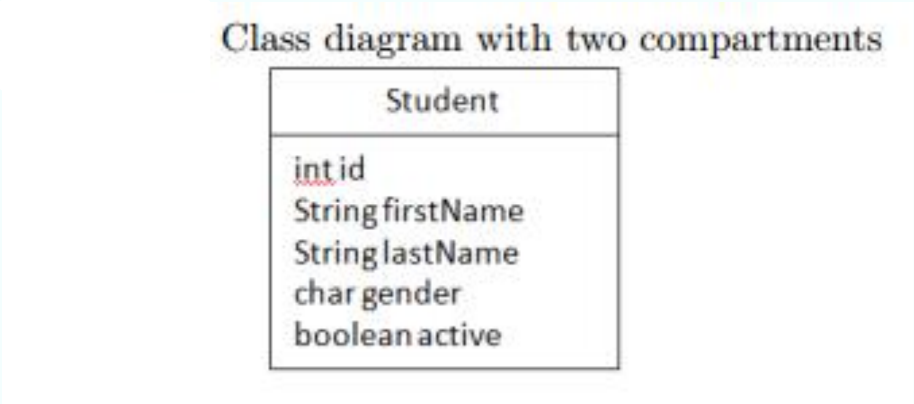
equals
Tests two student objects to see
if they are ‘equal’

Java Classes

Class is a template for objects

How are these shown in UML?

UML=unified modeling language



Class diagram with two compartments

Student

int id
String firstName
String lastName
char gender
boolean active

Objects

Objects:

- instantiated/created via `new` – lots of examples
- also called an *instance* – so we can speak of instance fields/methods
- How are these shown in UML?

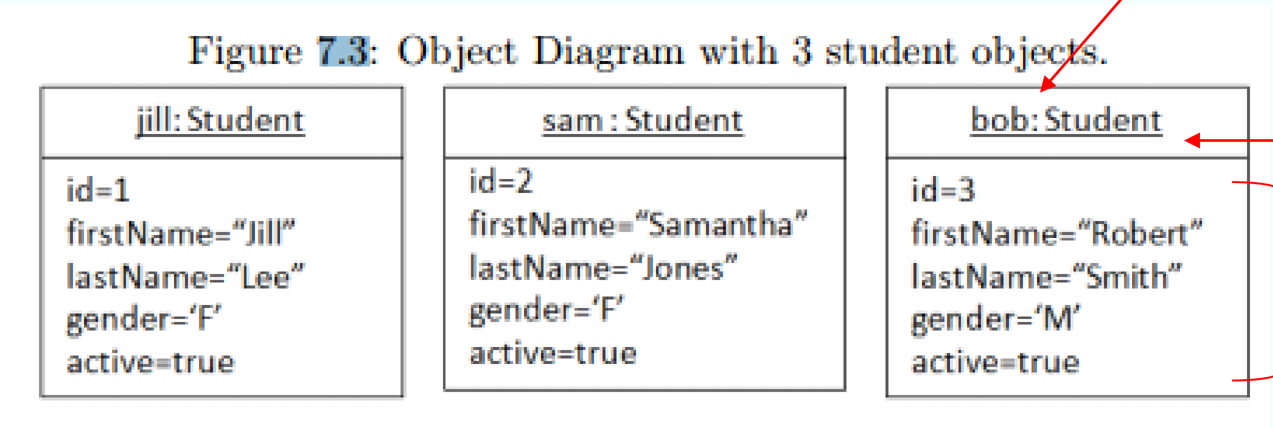


Figure 7.3: Object Diagram with 3 student objects.

jill: Student
id=1
firstName="Jill"
lastName="Lee"
gender='F'
active=true

sam : Student
id=2
firstName="Samantha"
lastName="Jones"
gender='F'
active=true

bob: Student
id=3
firstName="Robert"
lastName="Smith"
gender='M'
active=true

object name followed by ":" followed by class name

underlined

field values

Objects

Listing 7.1:

Creates two students

One using the no-arg constructors and setters

The other using a 4-arg constructor

Objects

```
/**
 * Create two student objects
 * using the two constructors
 */
public class UseConstructors
{
    public static void main (String[] args){
        // first, with the no-arg constructor
        Student jill = new Student();
        // use setters to complete the student object
        jill.setFirstName("Jill");
        jill.setLastName("Lee");
        jill.setGender('F');
        jill.setActive(true);
        // now with the other constructor
        Student sam = new Student("Samantha","Jones",'F',true);
        // display the students
        System.out.println(jill);
        System.out.println(sam);
    }
}
```

toString() is used automatically by JVM