* Object

Objects are **reference types**

We write Java code to create **classes**

We create **instances** of classes with the **new** keyword

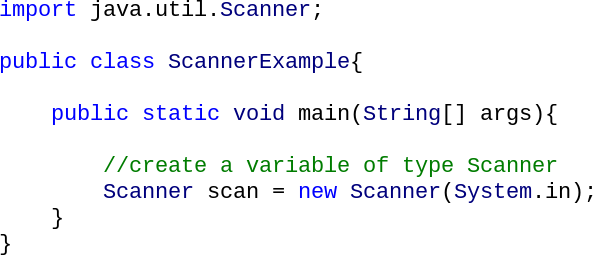
We may need to use an **import statement**

We **call methods** on **instances**

* Scan class

Note that we need import **java.util.Scanner;**

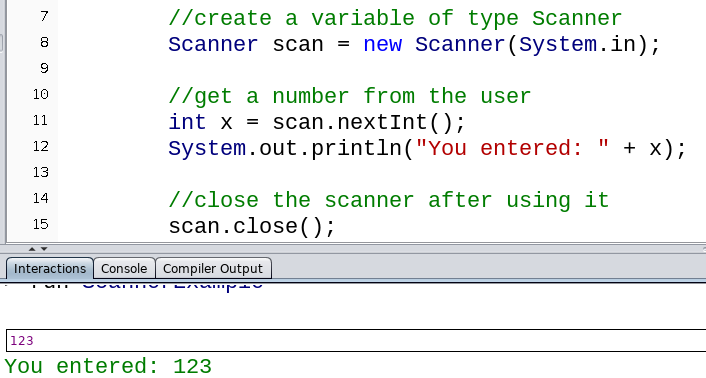
We create a variable scan with the type Scanner



Methods that can be called on a Scanner variable to get a value from the user.

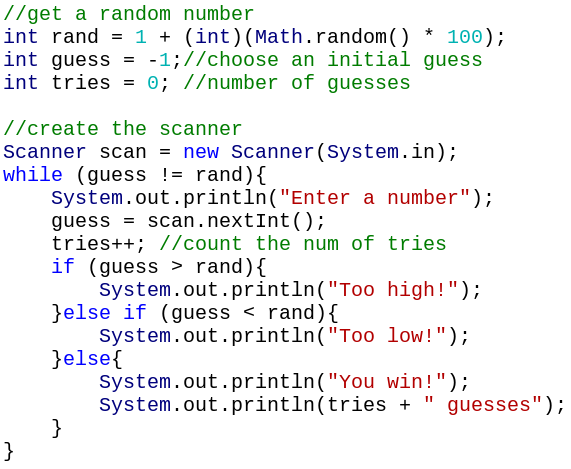
* int x = scan.nextInt()
* double d = scan.nextDouble()
* boolean b = scan.nextBoolean()
* String s = scan.next() ----a word as a String (HELLO)
* String line = scan.nextLine() ---- a whole line of input (HELLO WORLD)

These methods throw an **InputMismatchException** if the user enters the wrong type.



After using the Scanner, close it with the **close method**

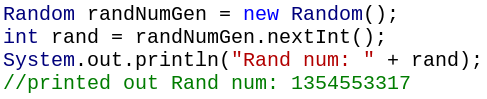
* Guessing game

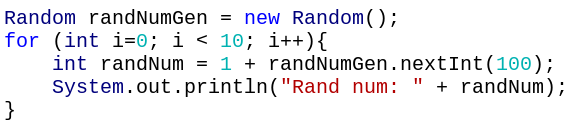


* Random Class

Note: We need to write import **java.util.Random** to use this class.

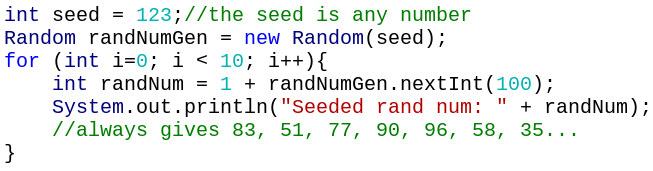
* Math.random() gives us a random integer from -2.3 billion to 2.3 billion



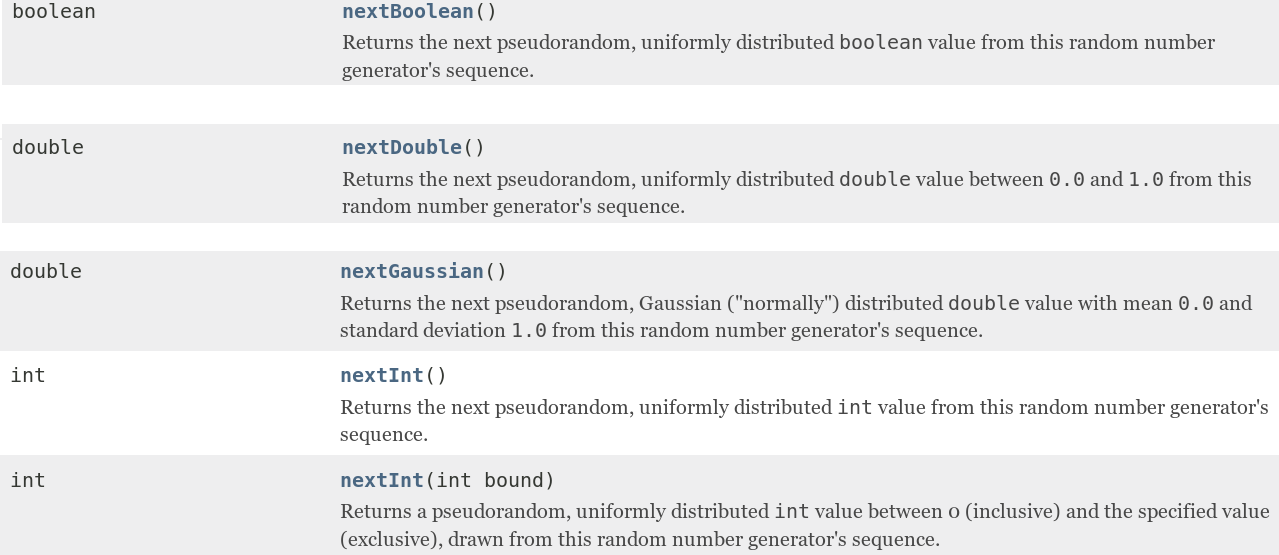
* Math.randInt(x) returns a random number between 0 and x (not including x) 
* Seeding----Get the same ten random numbers every time we ran the program.

Note that the seed isn’t the first random number

The seed just picks which sequence of random numbers should be generated



* Random Class Method



* Object-oriented programming

The Scanner and Random classes allow you to create **instances** and **call methods on those instances**

Object-oriented programming is about creating and using different classes and instances

A class is a collection of related methods and attributes

The purpose of object-orientated programming is to divide up your code into different components

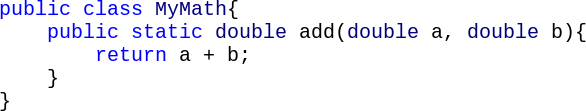
* Math class
* Attributes: Math.PI is a double, with value 3.141592653589793
* Methods: pow(double a, double b)
* Class method

---Methods in a classes

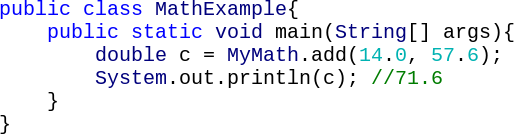
To call a method in a different class, we type the **class name, a period, and then the name of the method**

These classes must be in two different files in the **same directory**.

---Can access attributes in that class or other classes

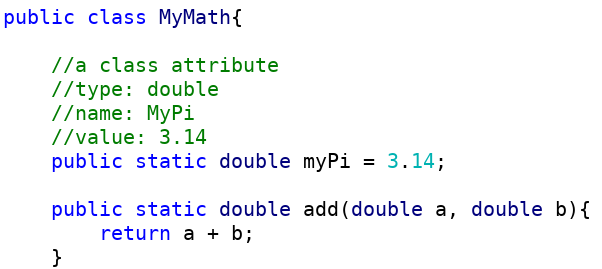


don’t need a main



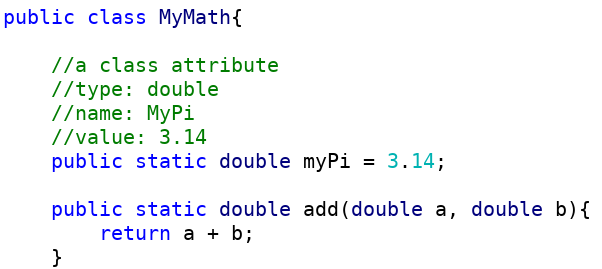
* Class attribute

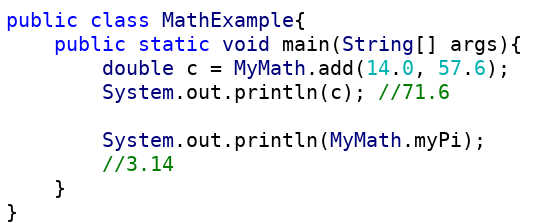
---Attributes belonging to the class

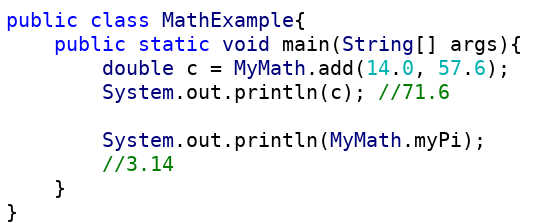


If you don’t provide a value to a class attribute, a **default value** will be given

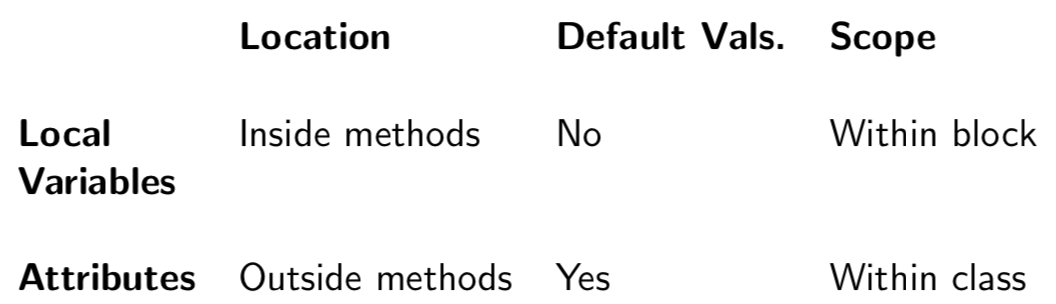
For primitive types: 0 or false For reference types: null



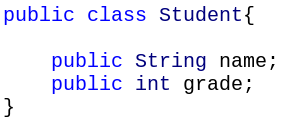




attribute vs variable



* Instances



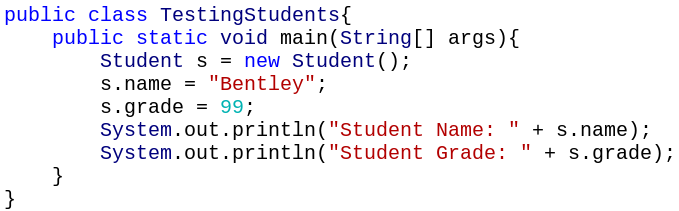
We’re creating a **new instance** of the Student class

Just like

Random rand = new java.util.Random();

int[] arr = new int[5];

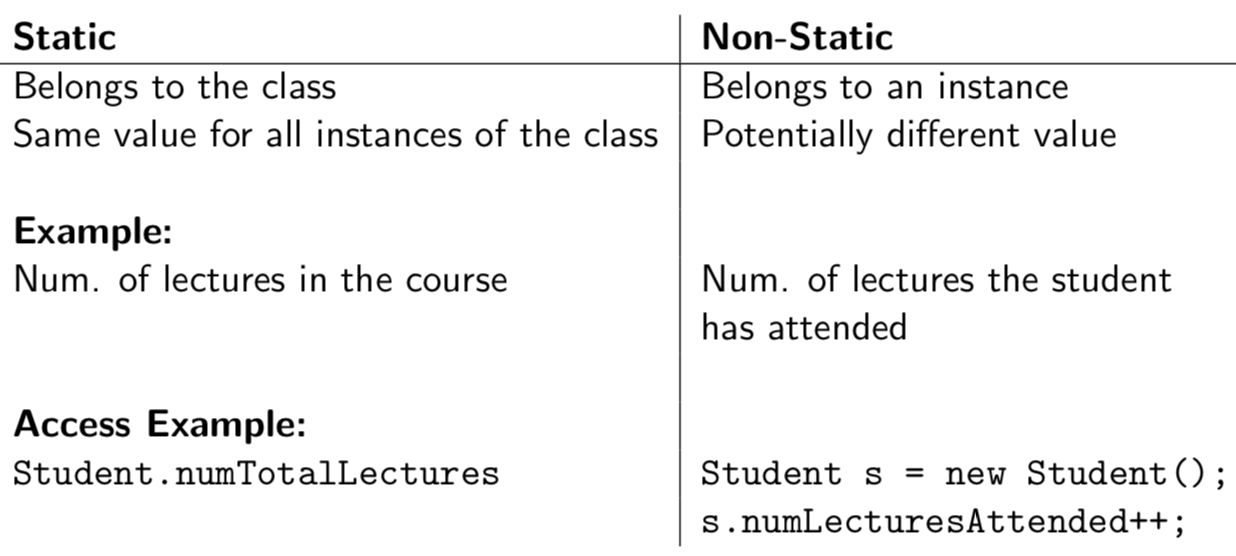
non-static



* Print-----------class name@address



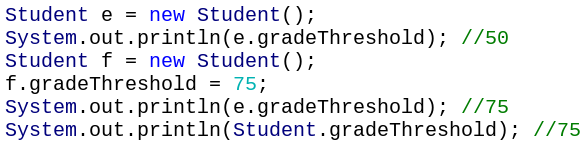
* Static/ Non-static attributes

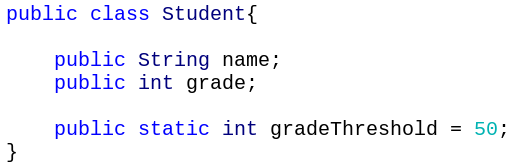


doesn’t mean non-changing

need an instance

Doesn’t make sense to write Student.grade = 76;

* change static attributes



* Static/ Non-static method

**Non-static methods can access static and non-static attributes/methods in the class**

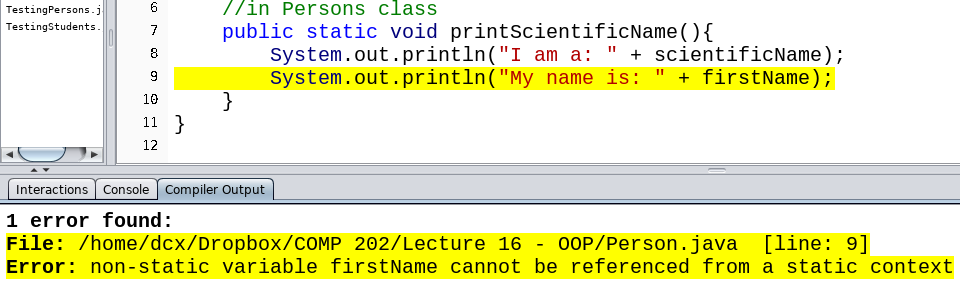
**Static methods can only access static attributes/methods in the class**

* First decide which attributes are non-static:

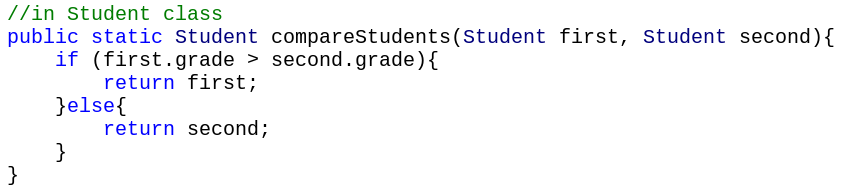
-----Does each instance has a different value for this attribute?

* Then decide if the method will access a non-static variable or method

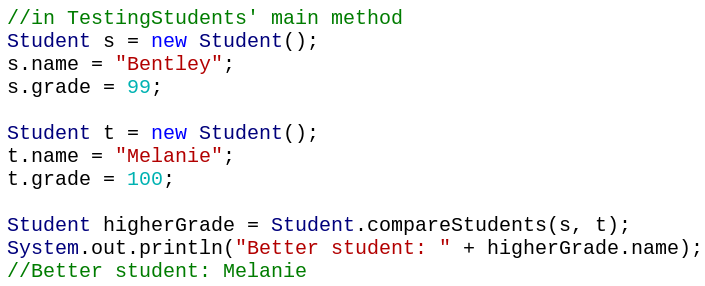
-----If the method accesses a non-static variable or method, it must be non-static



* instance as parameters



You cannot write grade in this method as it’s an non-static attribute

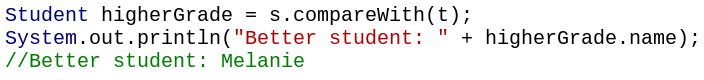


A static method can access non-static methods and attributes **if the method has access to an instance**

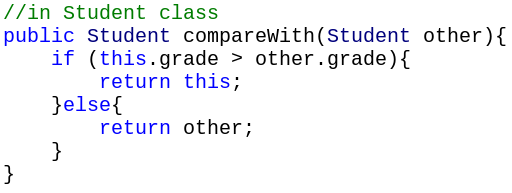
* this

In a non-static method, we might need to access the current instance

We can only use **this** in a non-static method

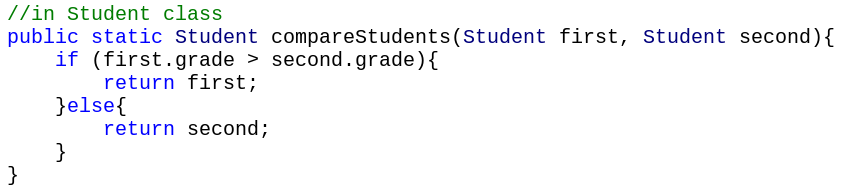


non-static version

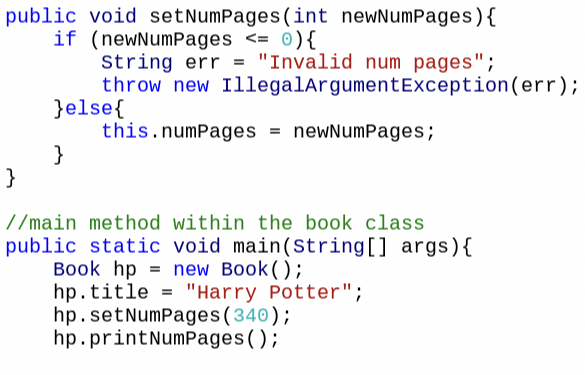
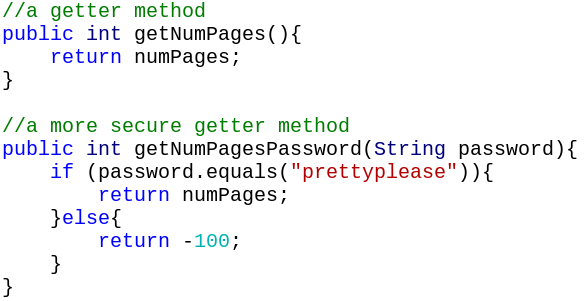


The **this** keyword is taking the place of the first parameter That’s because this refers to the current instance

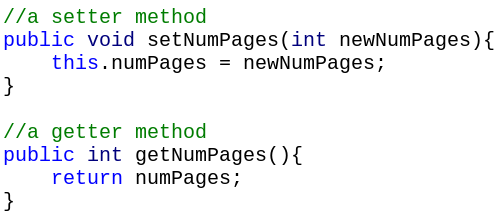
static version



* setter and getter

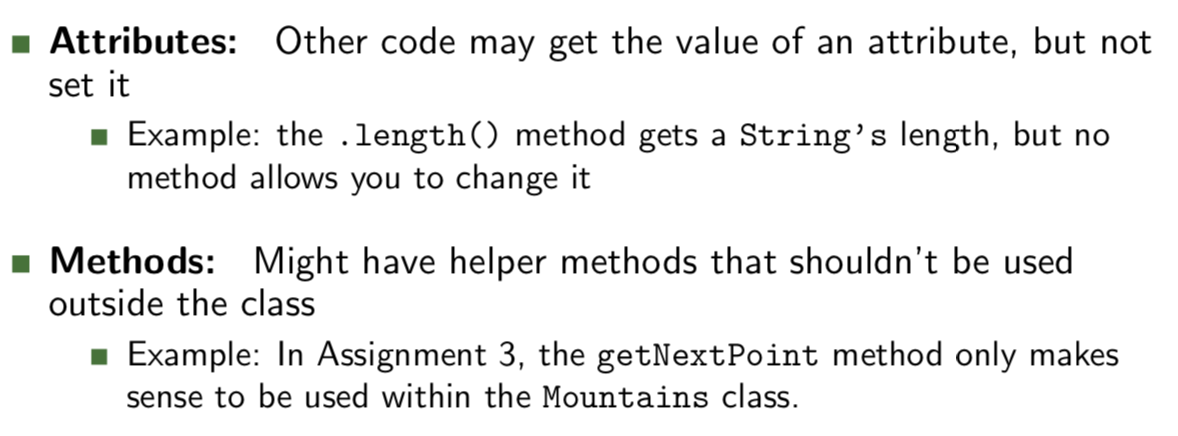
basic version

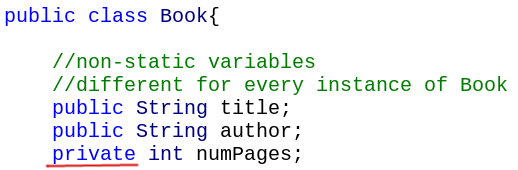


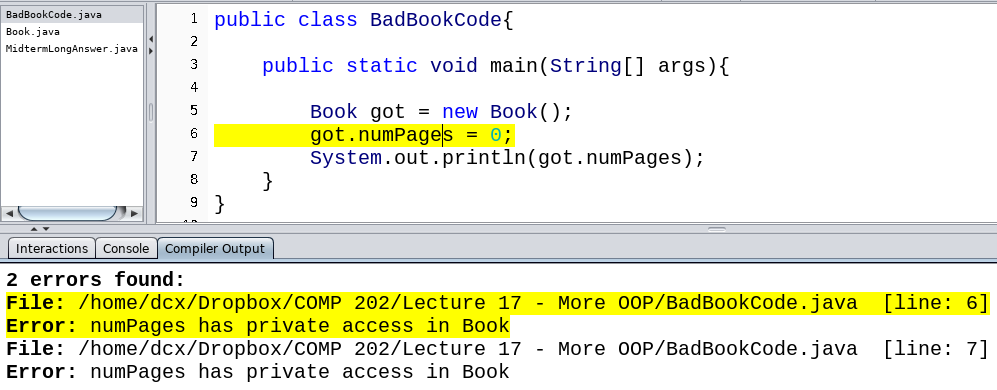
* public and private

If attributes or methods are private, they can’t be accessed outside the class

In the real world, you will not write every class or be able to change the source code The private keyword lets you control how others use the attributes and methods in your class







* Abstraction:

We should hide details, both to reduce complexity, and to allow changes to our code

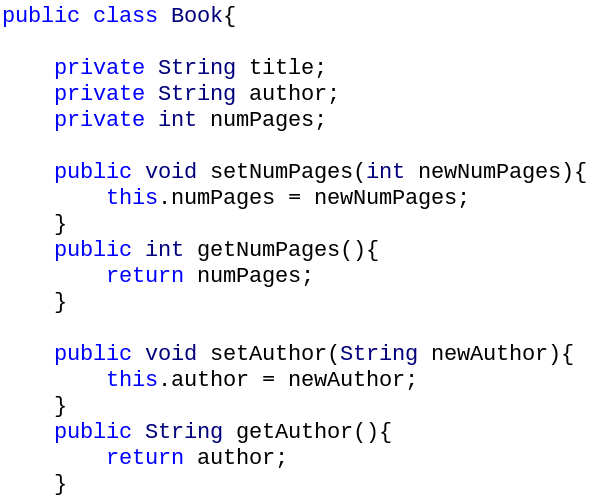
* General rules

Make all attributes in your class private if you can

Write getters and setters if other classes need to access these attributes

Make methods private, unless other classes need to access them

* Constructor example

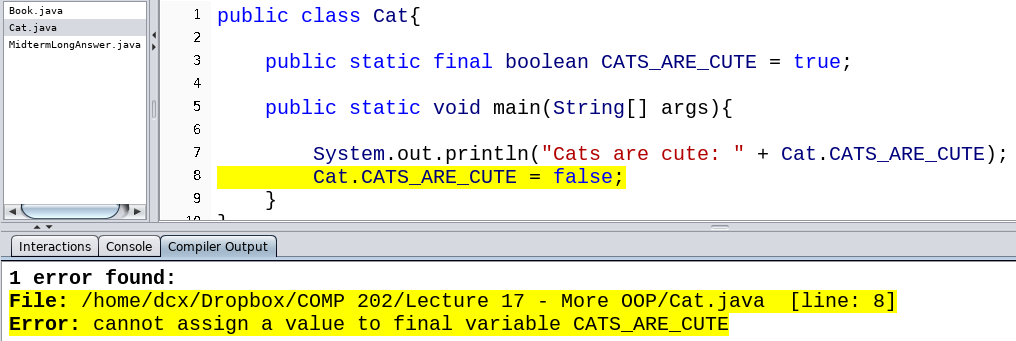


* final

If we create a final attribute, its value can never be changed after it has been initialized

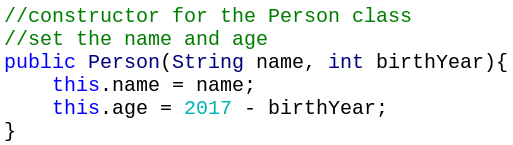
Ex: public static final int PASSING\_GRADE = 60;

We call static final variables **constants**-------name: all **uppercase, with underscores \_ between words**.



* Constrcutor

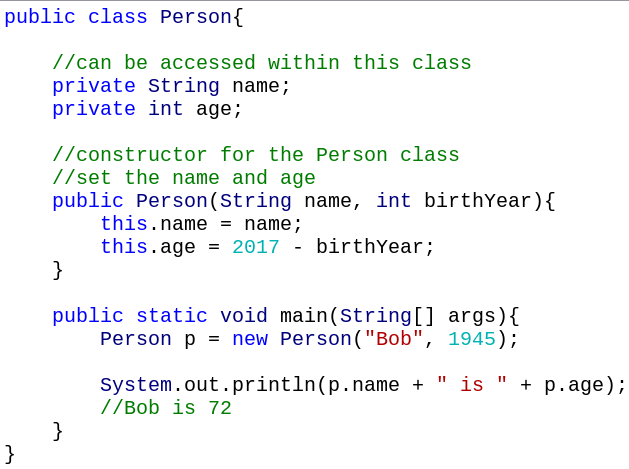
Constructors are executed when a new instance of a class is created. Constructors can be used to set attribute values



Name of the constructor method must be the same as the name of the class

No return type (not even void!)

Non-static method



When you have an attribute and a parameter with the same name, you need to put “this.” in front of the attribute.

If you do not write a constructor, the default constructor for a Person class looks like: 

If you write your own constructor, this will overwrite the default constructor!

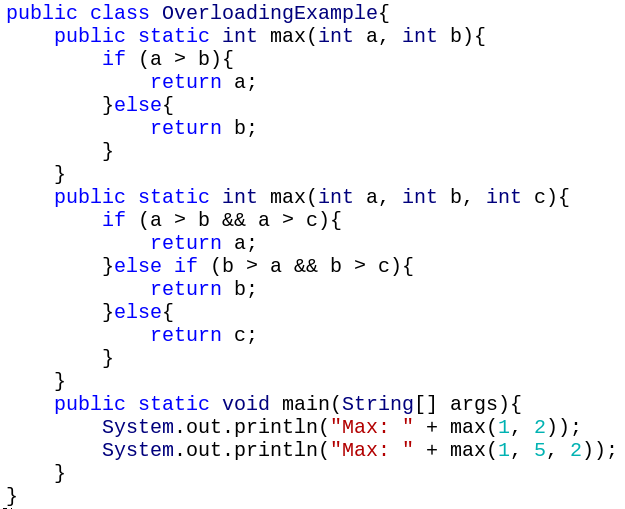
As in, you will then have to use the new constructor to create an instance

* Overloading

Might want two different methods in the **same class** to have the **same name** but **different parameters**

Ex: Changing a method’s algorithm depending on the types

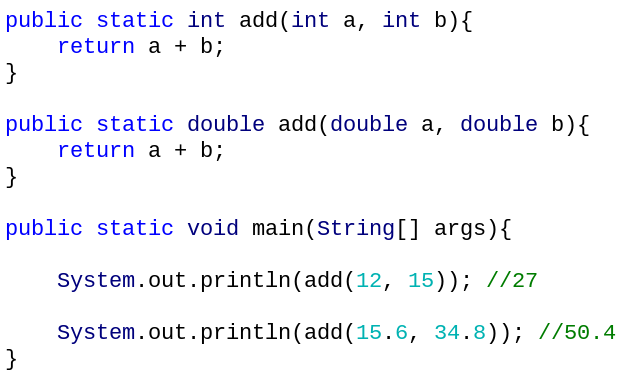
Different constructors



Java allows overloading based on the changes in method **parameters** (# and type)

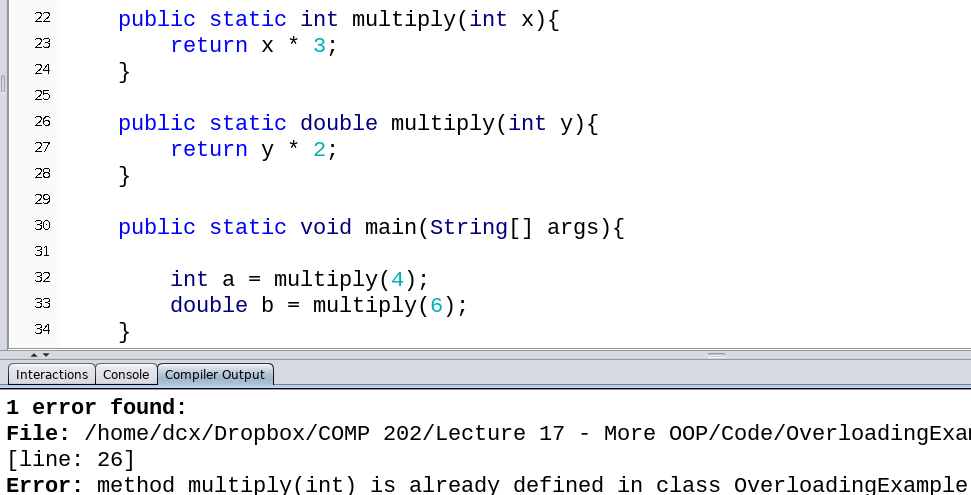
So public static int add(int i, int j) and public static double add(double a, double b) are okay

---Java automatically figures out when to call the int version, and when to call the double version



**Changing the return type and static/non-static of a method doesn’t allow overloading**

Java can’t tell which version of the method to call



public int add(int a, int b) public String add(int i, int j)

are called the same way, so this isn’t allowed

Ex: We have two ways to create a Random instance: with a seed, and without a seed



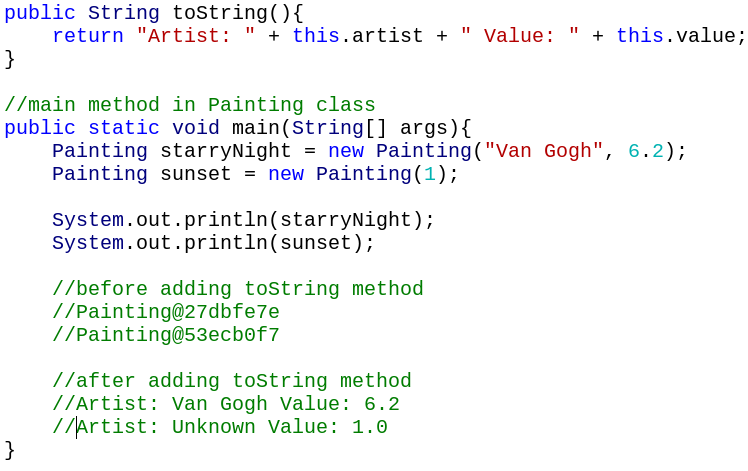
* toString()



When we print out an instance, we get its address

This is because the println method actually calls the toString method on the instance

The default code for the toString method is to print out the class name and address



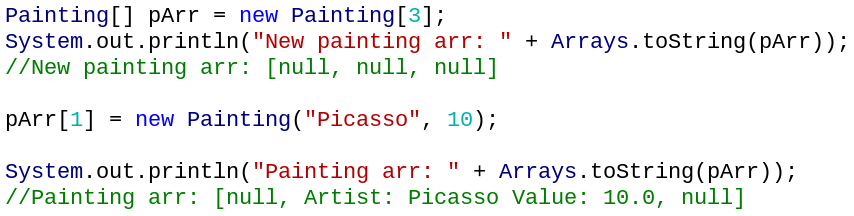
Returns a String when the instance is passed to a print method

Must have the header:

public String toString()

* storing instance

array of objects



Let’s put a Painting in an array

And use Arrays.toString()

Calls toString on each element in the array

