More About Methods

Christian A. Rodríguez

Edited by Juan Mendivelso

Object Oriented Programming

Defining Methods

Declaring methods

Parameters and arguments

Method signatures & Methods overloading

Declaring methods - Knowing services

Object A: You

Object B: Your pet

Knowing services

You need to know which of your pet's services (methods) you want your pet to perform.

- ✓ Sit
- ✓ Fetch
- ✓ Stay
- ✓ Dance

Declaring methods -Passing data

Object A: You

Object B: Your pet

Passing data

Depending on the service request, object you may need to give your pet some additional information so that your pet knows exactly how to proceed

- Fetch beer
- Fetch stick
- Fetch newspaper

Declaring methods - Expecting something?

Object A: You

Object B: Your pet

Expecting something?

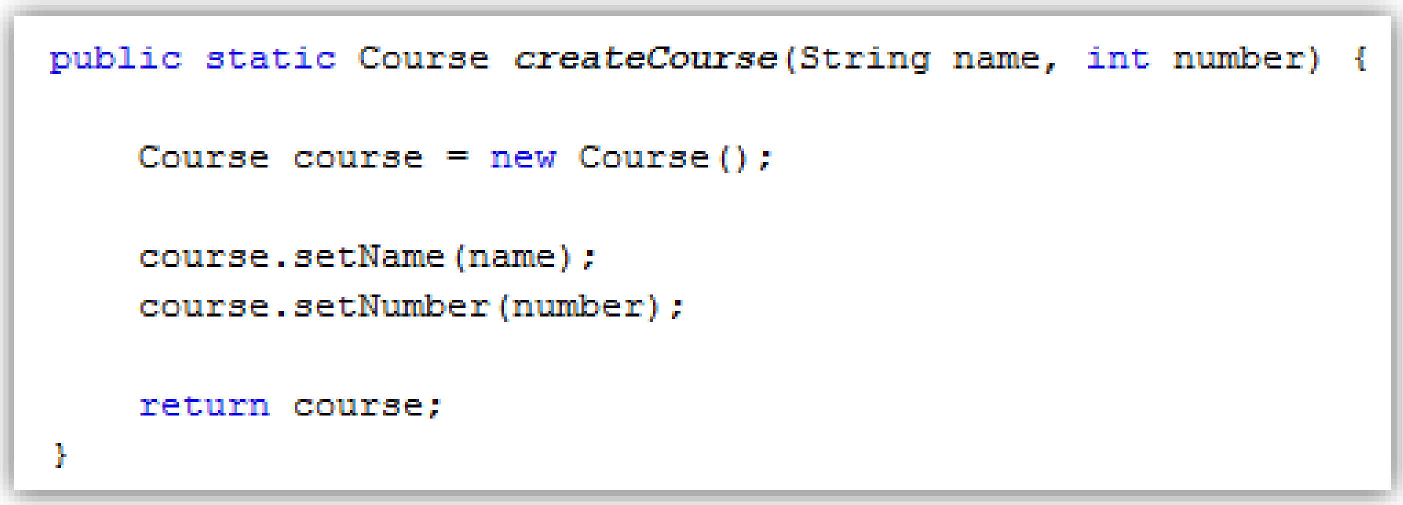
Your pet in turn needs to know whether you expects your pet to report back the outcome of what it has been asked to do.

- Are you expecting your pet give you the beer?
- Are you expecting your pet give you the stick
- Are you expecting your pet give you the newspaper?

Declaring methods - Java perspective

Knowing services?

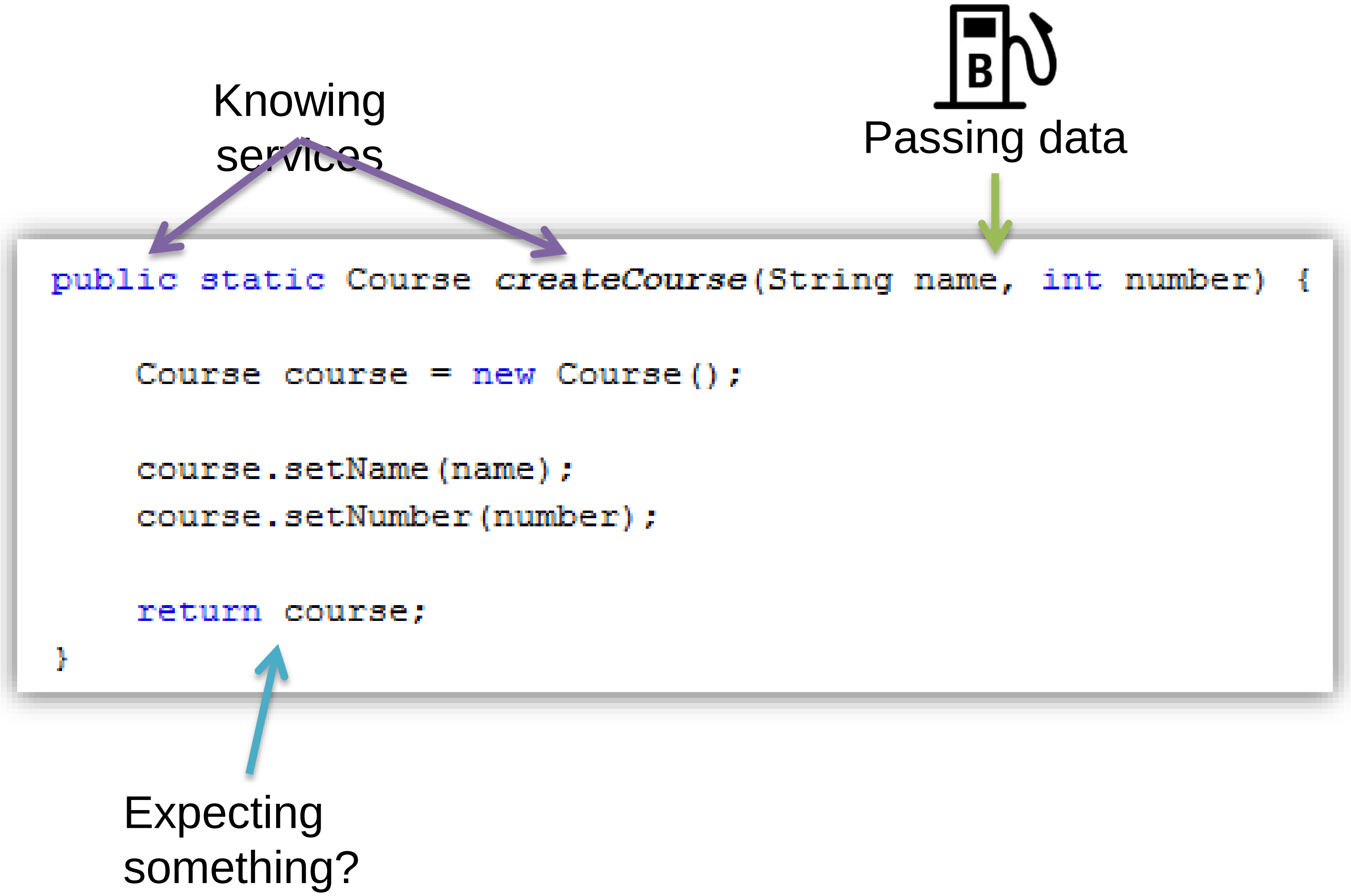
Passing data?



```
public static Course createCourse(String name, int number) {

    Course course = new Course();

    course.setName(name);
    course.setNumber(number);

    return course;
}
```

Expecting something?

Declaring methods in Java terms



```
public static Course createCourse(String name, int number) {

    Course course = new Course();

    course.setName(name);
    course.setNumber(number);

    return course;
}
```

Passing data

How it is the **passing** data process?

Parameters and arguments

Parameter are considered **local variables** in the method.

Parameter

```
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

Argument are refered to **values.**

```
Student student = new Student();
student.setFirstName("Bruce Wayne");
```

Argument

Parameters and arguments

When a method is called, each parameter is initialized with the corresponding argument passed.

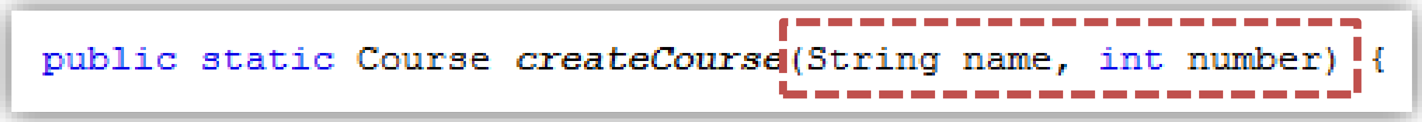
Parameter

```
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

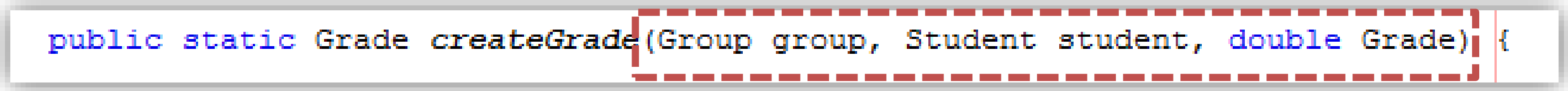
```
Student student = new Student();
student.setFirstName("Bruce Wayne");
```

Argument

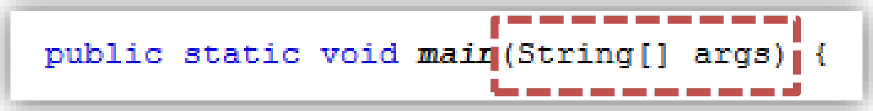
Parameters examples



```java
public static Course createCourse(String name, int number) {
```

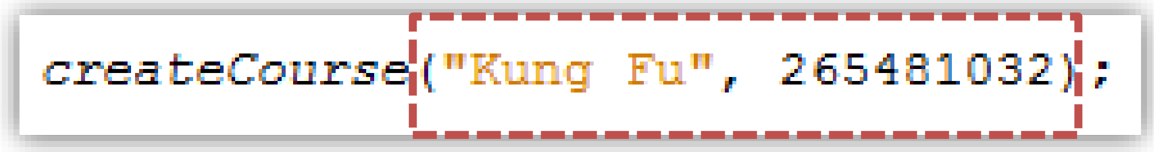


```java
public static Grade createGrade(Group group, Student student, double Grade) {
```

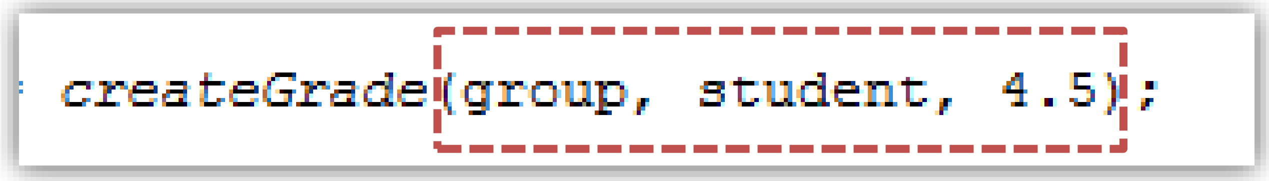


```java
public static void main(String[] args) {
```

Arguments examples



```
createCourse("Kung Fu", 265481032);
```

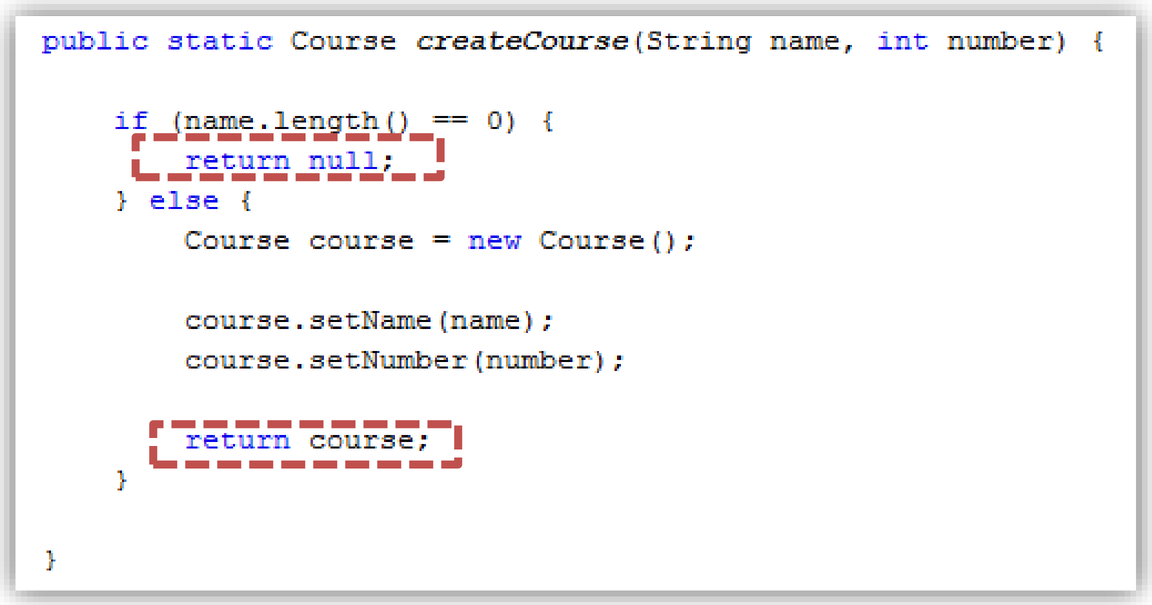


```
createGrade(group, student, 4.5);
```

Returning data

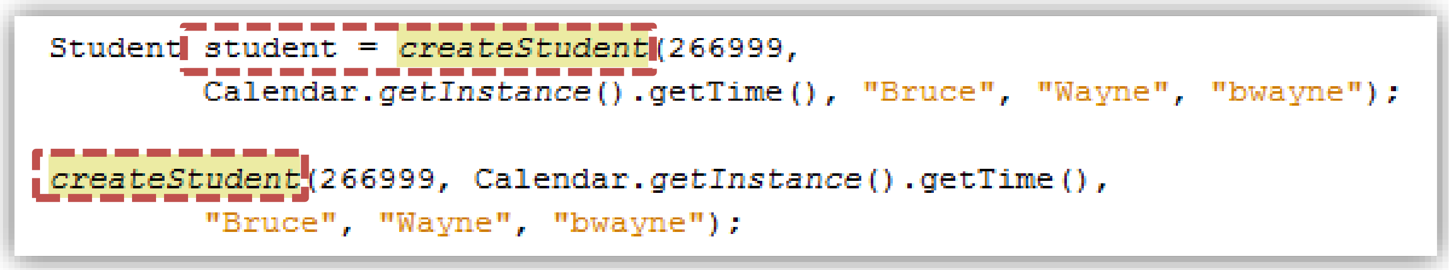
How it is the **returning** data process?

Methods can define zero or many returning points



```
public static Course createCourse(String name, int number) {

    if (name.length() == 0) {
        return null;
    } else {
        Course course = new Course();

        course.setName(name);
        course.setNumber(number);

        return course;
    }

}
```

Clients can use or not returning data

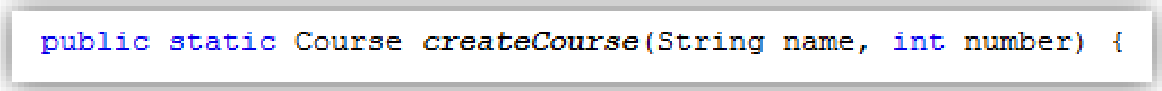


```
Student student = createStudent(266999,
        Calendar.getInstance().getTime(), "Bruce", "Wayne", "bwayne");

createStudent(266999, Calendar.getInstance().getTime(),
        "Bruce", "Wayne", "bwayne");
```

Method header

- Method header
  - Method's Name
  - Method's return type
  - List of parameters with type and names

Java definition for a method



```java
public static Course createCourse(String name, int number) {
```

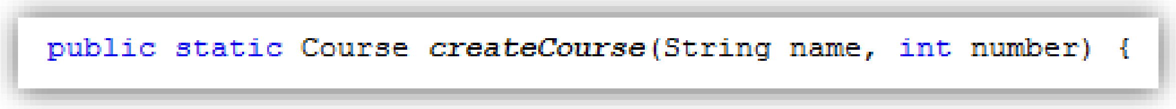
This method header is:

```java
Course createCourse(String name, int number)
```

Method signatures

Methods have signatures which indicates

- Method's Name
- Order, types and number of parameters



```
public static Course createCourse(String name, int number) {
```

This Method signature is:

`createCourse(String , int)`

The method **createCourse** declares **two** parameters of type **String** and **int** respectively

Method signature is unique

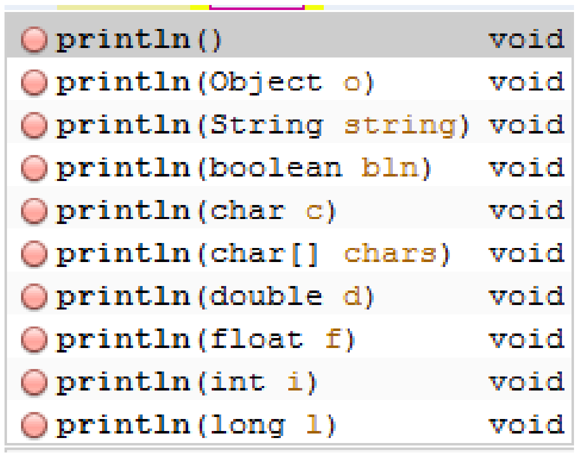
Methods overloading

Methods from the **same class** can be offered with a **unique name** but with **different signature**

println overloading

```
public void println() {
```

```
public void println(char[] chars) {
```



```
println()                void
println(Object o)        void
println(String string)   void
println(boolean bln)     void
println(char c)          void
println(char[] chars)    void
println(double d)        void
println(float f)         void
println(int i)           void
println(long l)          void
```

```
public void println(long l) {
```

```
public void println(String string) {
```

```
public void println(boolean bln) {
```

```
public void println(char c) {
```

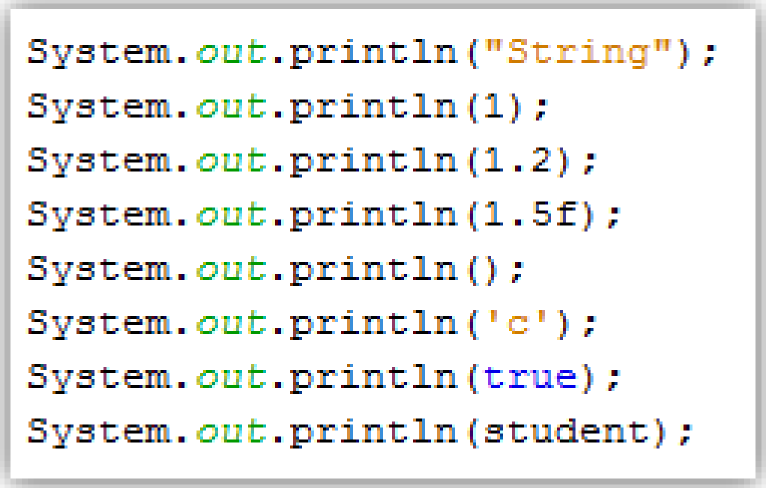
```
public void println(int i) {
```

```
public void println(double d) {
```

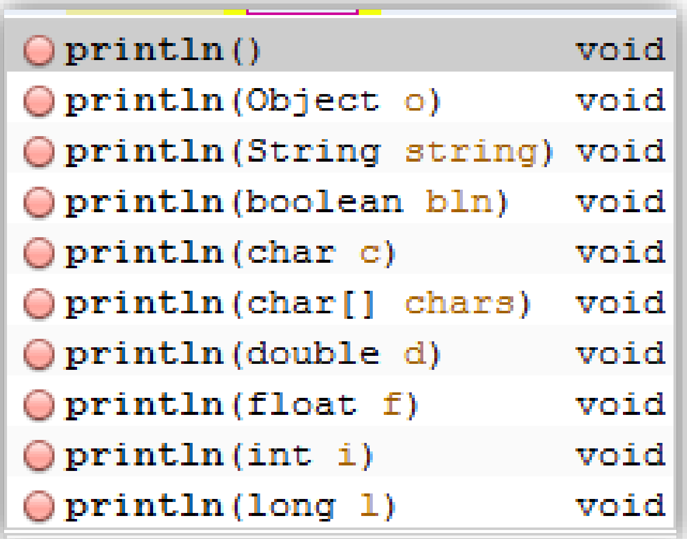
```
public void println(float f) {
```

```
public void println(Object o) {
```

println overloading

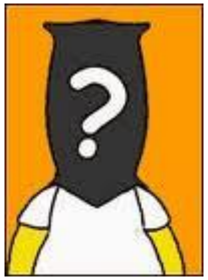


```
System.out.println("String");
System.out.println(1);
System.out.println(1.2);
System.out.println(1.5f);
System.out.println();
System.out.println('c');
System.out.println(true);
System.out.println(student);
```



```
println()                void
println(Object o)        void
println(String string)   void
println(boolean bln)     void
println(char c)          void
println(char[] chars)    void
println(double d)        void
println(float f)         void
println(int i)           void
println(long l)          void
```

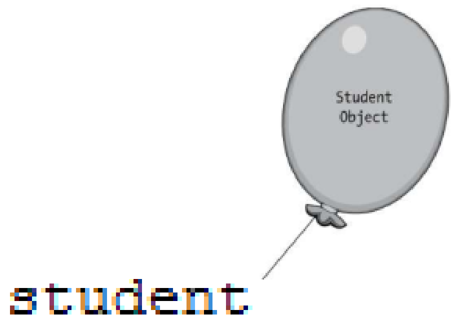
Compiler choose the correct method checking the types in the list of arguments passed to parameters.



Constructors

Do you remember how to instantiate?

```
Student student = new Student();
```



Constructors

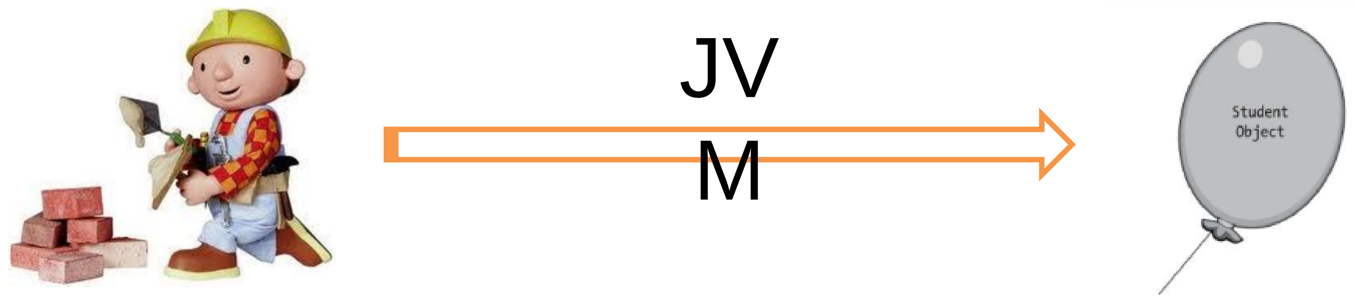
This is the Student class constructor

```
Student student = new Student();
```

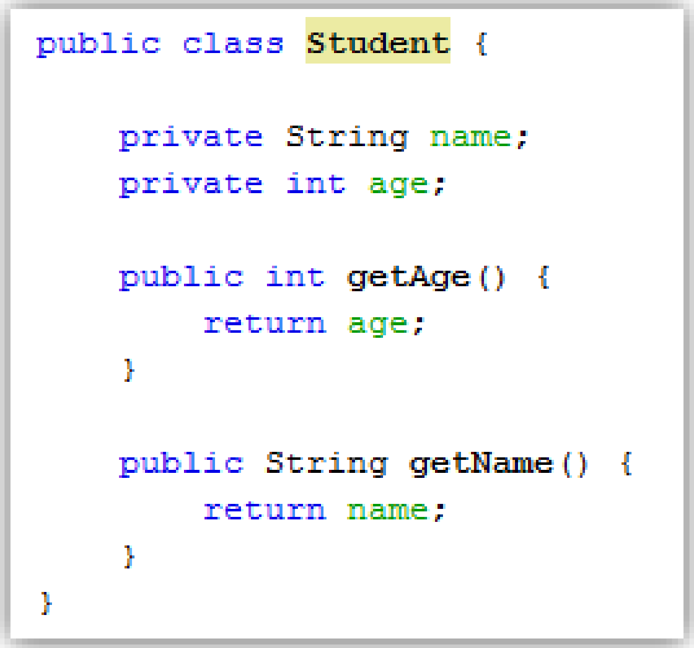
Invoking a constructor serves as a request to the JVM to <u>construct</u> (instantiate) a brand-new object

Constructors

Constructors are special type of procedures which **are responsible to ask the JVM to inflate a new helium balloon**

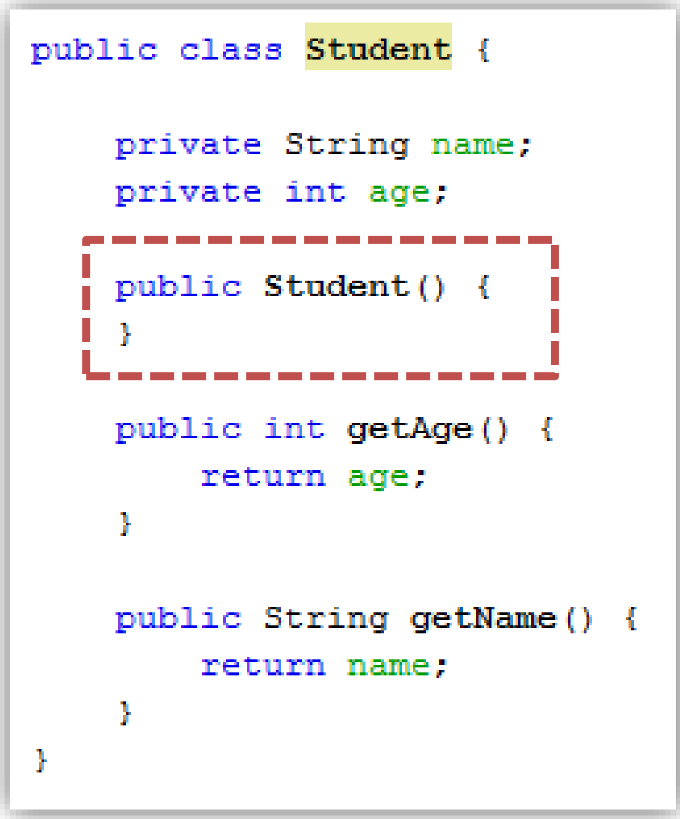


Default constructor



```
public class Student {

    private String name;
    private int age;

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }
}
```

=

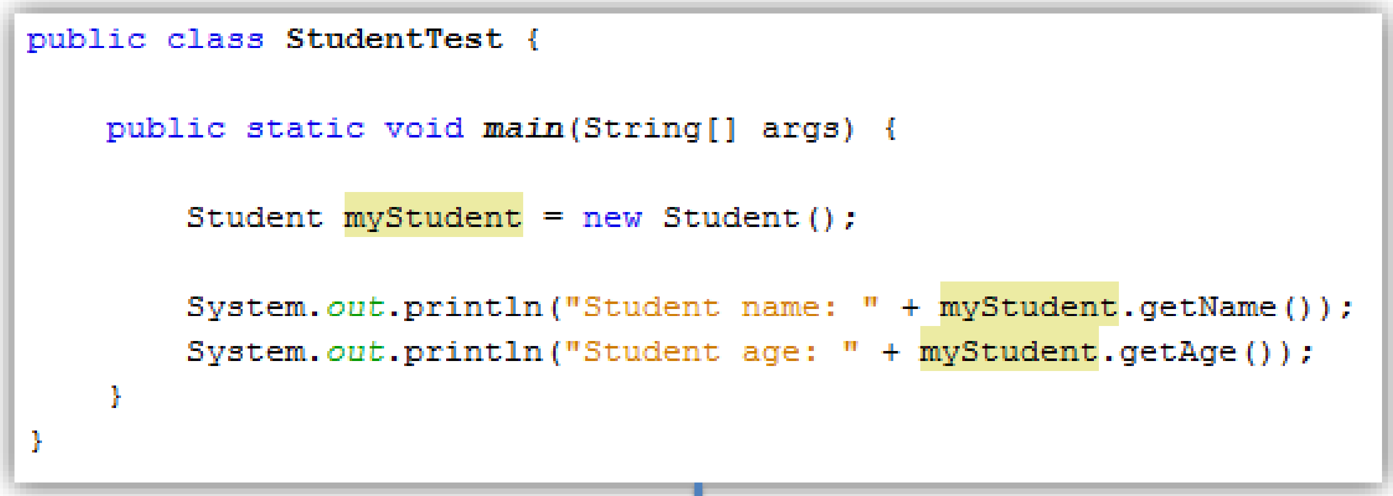


```
public class Student {

    private String name;
    private int age;

    public Student() {
    }

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }
}
```

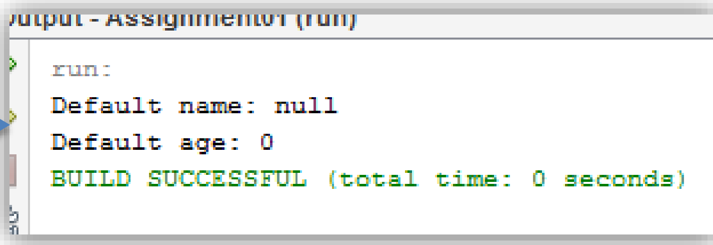
If there is no defined constructors, the JVM
use the by **default constructor**

Default constructor

Default constructors set all attributes to their zero-equivalent default values



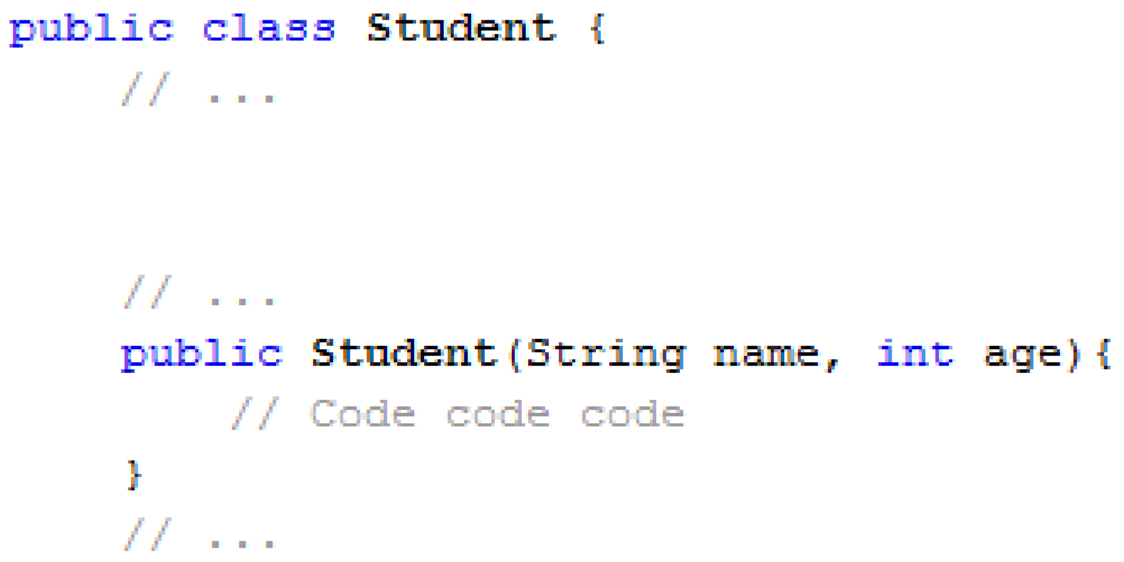
```
public class StudentTest {

    public static void main(String[] args) {

        Student myStudent = new Student();

        System.out.println("Student name: " + myStudent.getName());
        System.out.println("Student age: " + myStudent.getAge());
    }
}
```



utput - Assignment01 (run)

```
run:
Default name: null
Default age: 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Explicit constructors

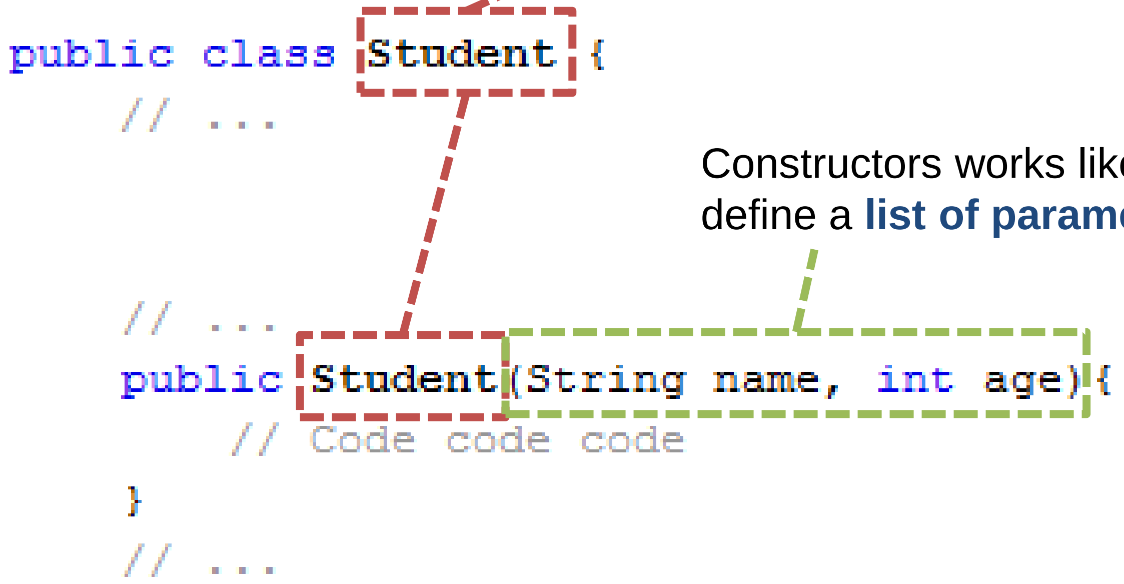


```
public class Student {
    // ...



    // ...
    public Student(String name, int age){
        // Code code code
    }
    // ...
```

We use **explicit constructors** if we wish to do something more “interesting” to initialize an object when it is first instantiated

Explicit constructors rules

Constructor's name **must be exactly the same** as the name of the class for which we're writing the constructor

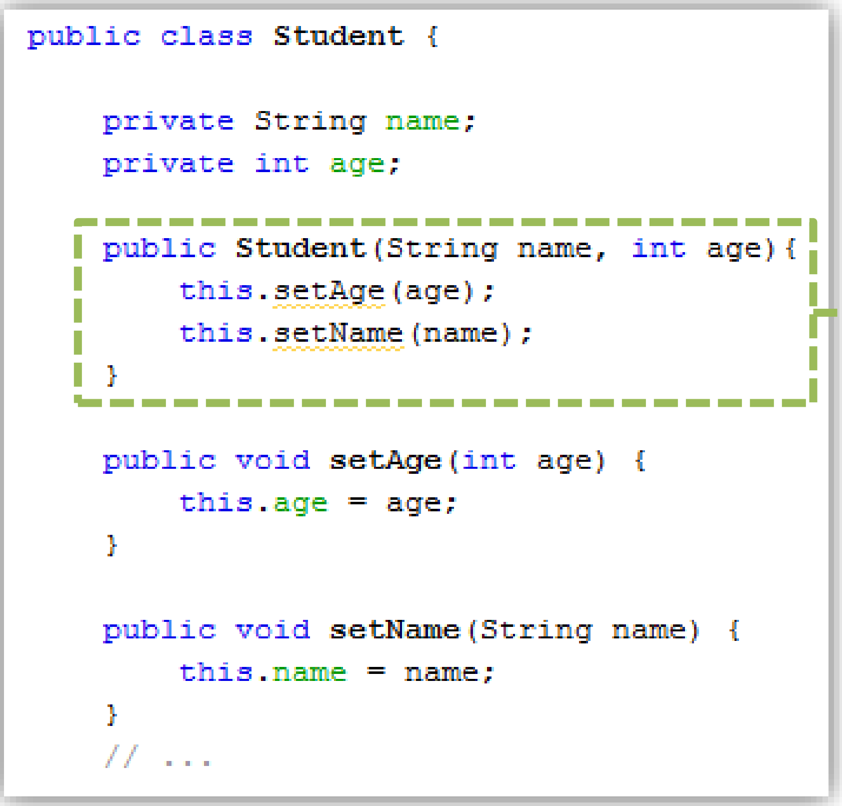


```
public class Student {
    // ...


    // ...
    public Student(String name, int age){
        // Code code code
    }
    // ...
```

Constructors works like another method, can define a **list of parameters**

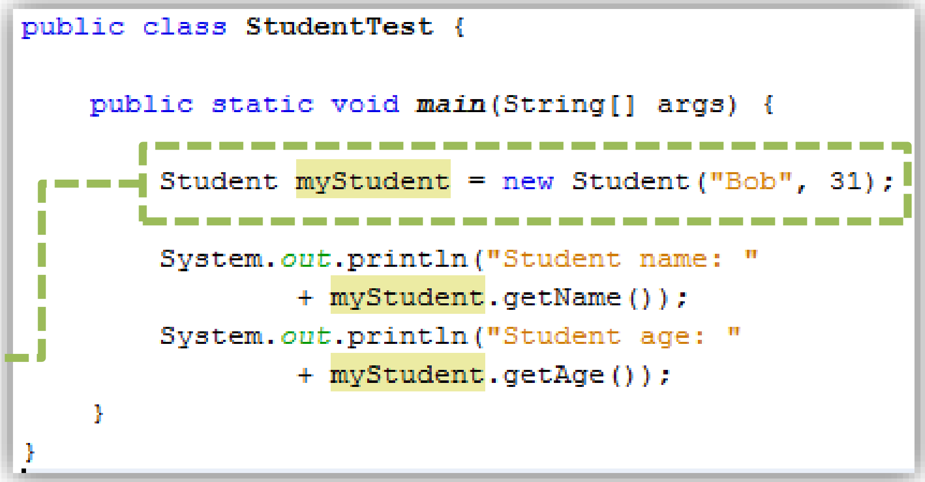
We **cannot specify a return type** for a constructor; by definition, a constructor returns a **reference to a newly created object of the class type**

Passing parameters to constructors



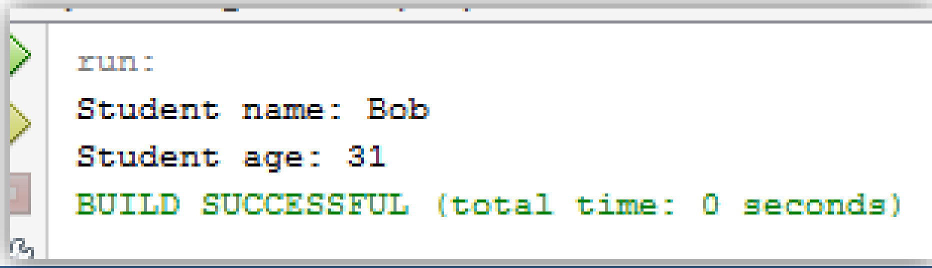
```
public class Student {

    private String name;
    private int age;

    public Student(String name, int age){
        this.setAge(age);
        this.setName(name);
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }
    // ...
```

Class definition



```
public class StudentTest {

    public static void main(String[] args) {

        Student myStudent = new Student("Bob", 31);

        System.out.println("Student name: "
                + myStudent.getName());
        System.out.println("Student age: "
                + myStudent.getAge());
    }
}
```

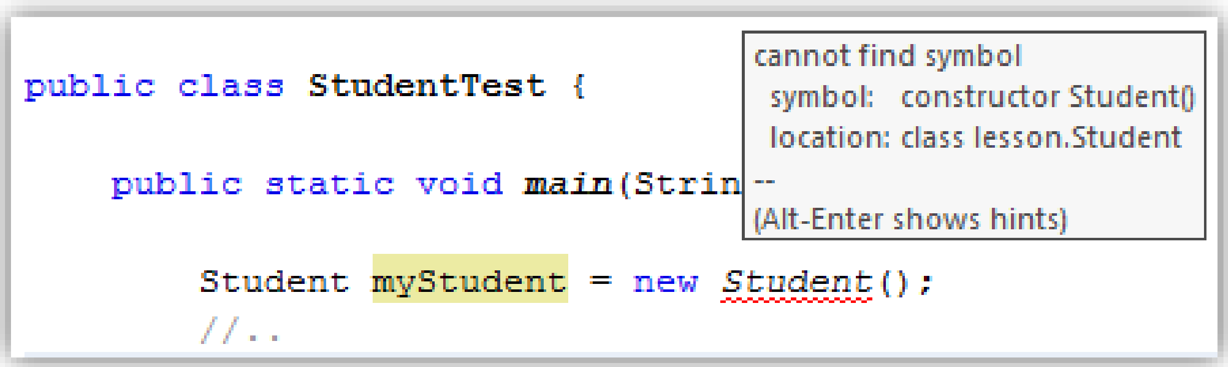
Test Class



```
run:
Student name: Bob
Student age: 31
BUILD SUCCESSFUL (total time: 0 seconds)
```

Be careful

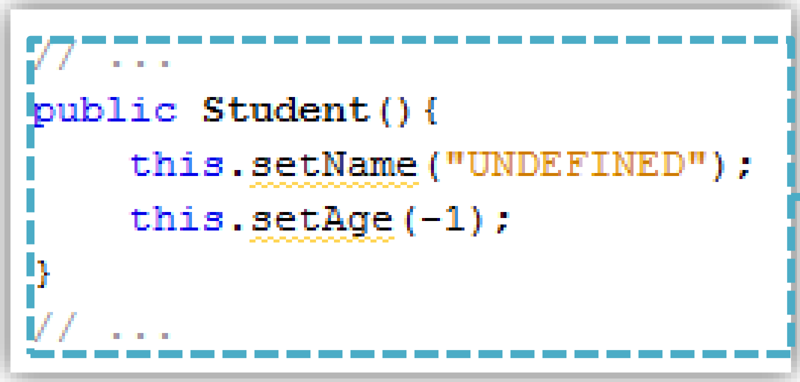
If there is at least one constructor defined by us, we **cannot** use the default constructor



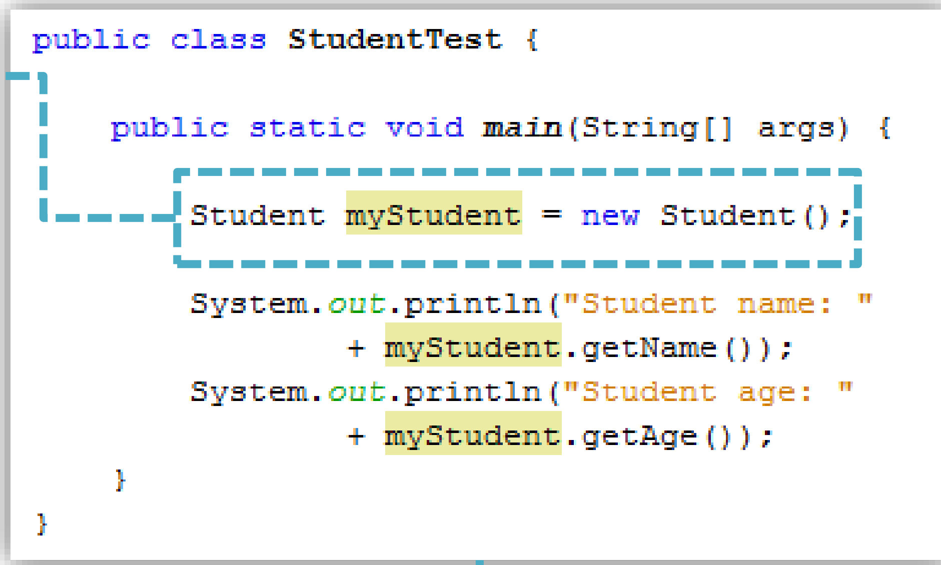
```
public class StudentTest {

    public static void main(Strin

        Student myStudent = new Student();
        //..
```

```
cannot find symbol
  symbol:   constructor Student()
  location: class lesson.Student
--
(Alt-Enter shows hints)
```

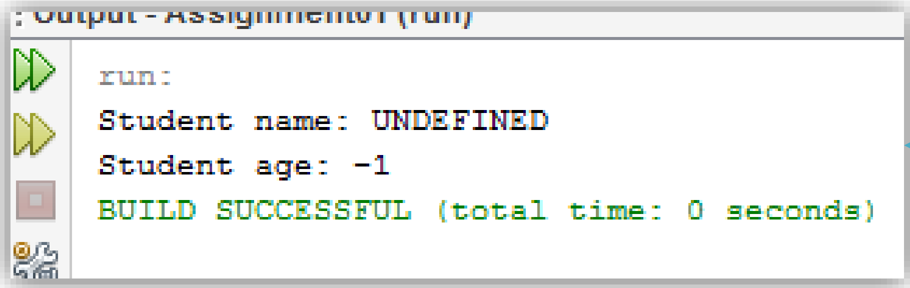
Replacing the Default Parameterless Constructor



```
// ...
public Student(){
     this.setName("UNDEFINED");
     this.setAge(-1);
}
// ...
```



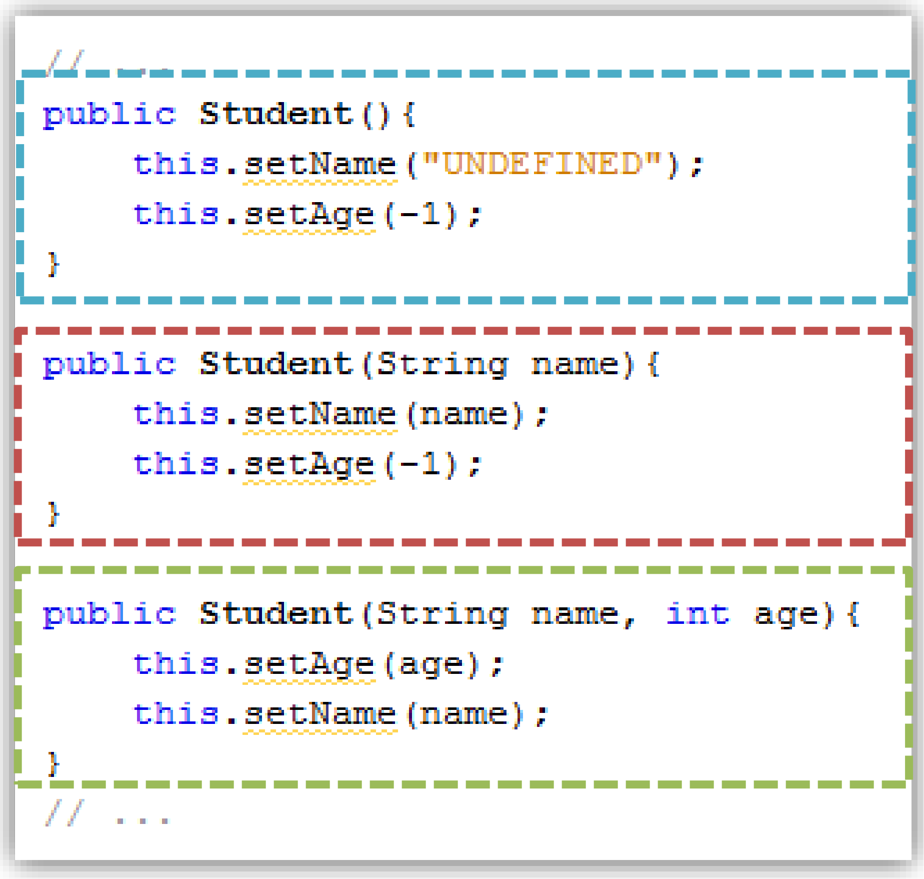
```
public class StudentTest {

    public static void main(String[] args) {
        Student myStudent = new Student();
        System.out.println("Student name: "
                + myStudent.getName());
        System.out.println("Student age: "
                + myStudent.getAge());
    }
}
```



```
run:
Student name: UNDEFINED
Student age: -1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Overloading Constructors

It is possible to overload Constructors like any other method



```
// ...
public Student(){
     this.setName("UNDEFINED");
     this.setAge(-1);
}
public Student(String name){
     this.setName(name);
     this.setAge(-1);
}
public Student(String name, int age){
     this.setAge(age);
     this.setName(name);
}
// ...
```

Constructor 1 signature

`Student ( )`

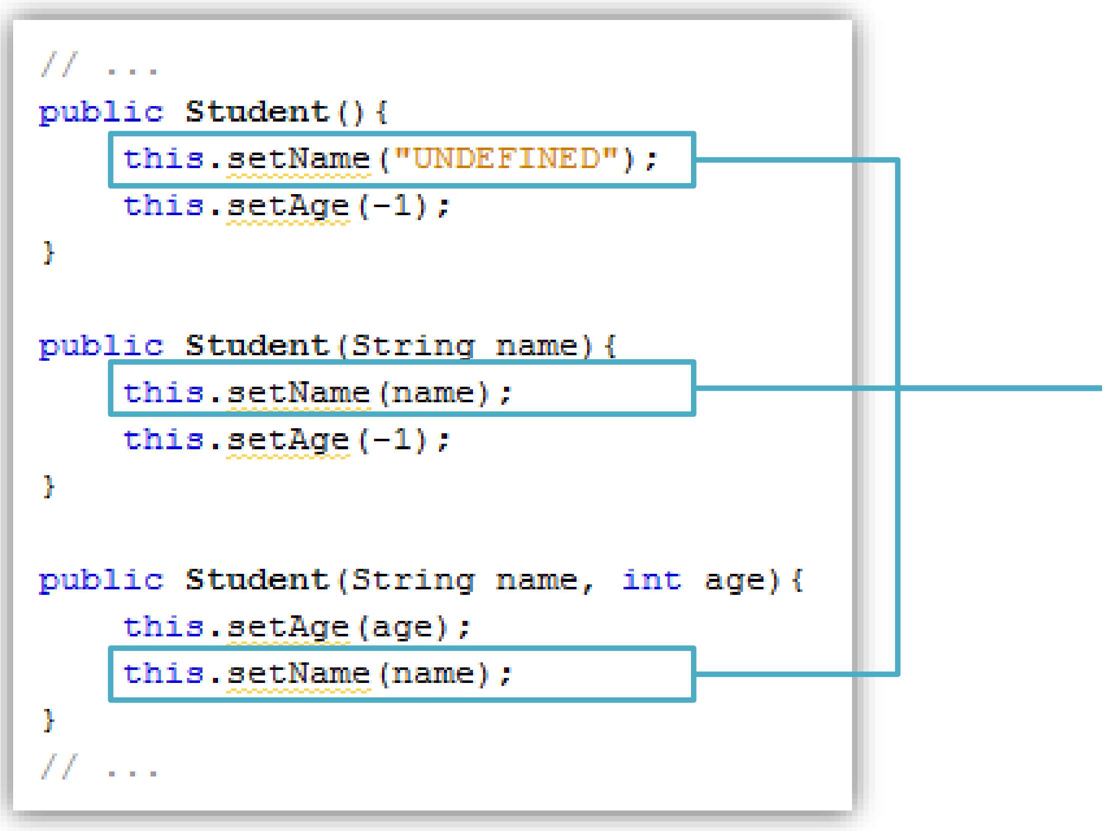
Constructor 2 signature

`Student ( String )`

Constructor 3 signature

`Student ( String , int )`

Constructors reuse



```
// ...
public Student(){
    this.setName("UNDEFINED");
    this.setAge(-1);
}

public Student(String name){
    this.setName(name);
    this.setAge(-1);
}

public Student(String name, int age){
    this.setAge(age);
    this.setName(name);
}
// ...
```

Code duplication

Constructors reuse

It is possible to reuse Constructors using the keyword **this**

```java
// ...
public Student(){
    this.setName("UNDEFINED");
    this.setAge(-1);
}

public Student(String name){
    this.setName(name);
    this.setAge(-1);
}

public Student(String name, int age){
    this.setAge(age);
    this.setName(name);
}
// ...
```

=

```java
// ...
public Student() {
    this("UNDEFINED", -1);
}

public Student(String name) {
    this(name, -1);
}

public Student(String name, int age) {
    this.setAge(age);
    this.setName(name);
}
// ...
```

Reusing constructors can avoid duplication of code

References

- [Barker] J. Barker, *Beginning Java Objects: From Concepts To Code*, Second Edition, Apress, 2005.