Java

R2.01 : Object-oriented development (OOD)

<u>Coordinator : Isabelle Blasquez</u>
My name: Cristina Onete
cristina.onete@gmail.com

Today's main goal

Learn to write basic Java code for a basic application

Concepts	Java
❖ Classes, attributes	❖ Variable types, primitive/non-primitive types, public vs. private ❖ Syntax: classes and attributes
❖ Instantiation, objects, constructors	❖ Basic variable manipulation ❖ Constructor syntax and class instantiation
❖ Multiple classes, main method	❖ Main method syntax ❖ Re : public vs. private variables ❖ Using methods & attributes outside class ❖ The String toString() method

Java: short history

- 1991 : James Gosling, Mike Sheridan, Patrick Naughton embark on the quest of developing Java
- 1995 : Sun Microsystems adheres to the "Write Once Run Anywhere" paradigm : a reference implementation of Java by Sun
- 1998-1999 : Java 2 released, including J2EE (today Jakarta EE) for distri-buted computing/web services; J2ME for mobile applications

J2 SE (standard) | **J2 EE (entreprise)** | **J2 ME (micro)**

- 2007 : Java makes its code open-source (GNU GPL license)
- 2010 : Oracle buys Java. Today, Java is all around us.

Java's main design goals

source: Design Goals of the Java programming language, Oracle 1999

- Simple, object-oriented, and familiar
- Robust and secure
- Architecture-neutral and portable
- It must execute with high performance
- Interpreted, threaded, and dynamic

Is Java different from other programming languages ?

Java vs. C and C++

- Imperative language (C, C++)
  - Relies on functions and procedures
  - Programs consisting of function definitions and function calls
  - Each function caracterised by "signature": I/O types, name
  - Local and global variables

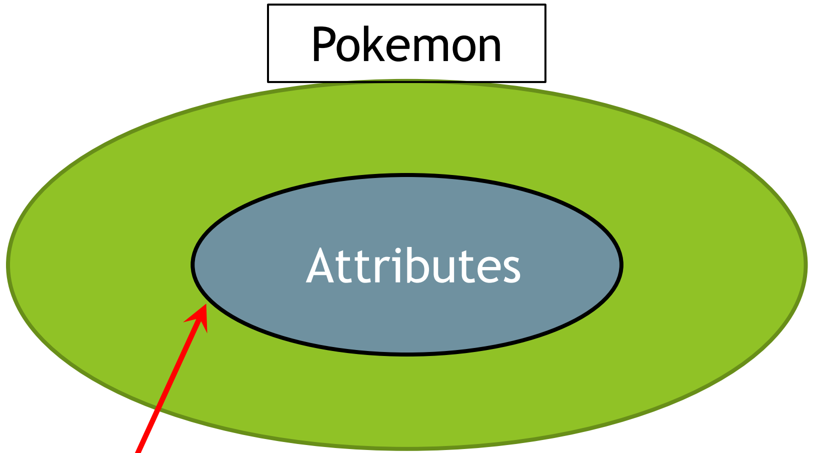
- Object-oriented language (Java)
  - Object oriented, using classes
  - Objects instantiate classes; they have their own attributes and methods
  - Methods caracterised by signatures, associated to classes
  - All variables local (to methods, classes, etc.)

Java is also verbose !

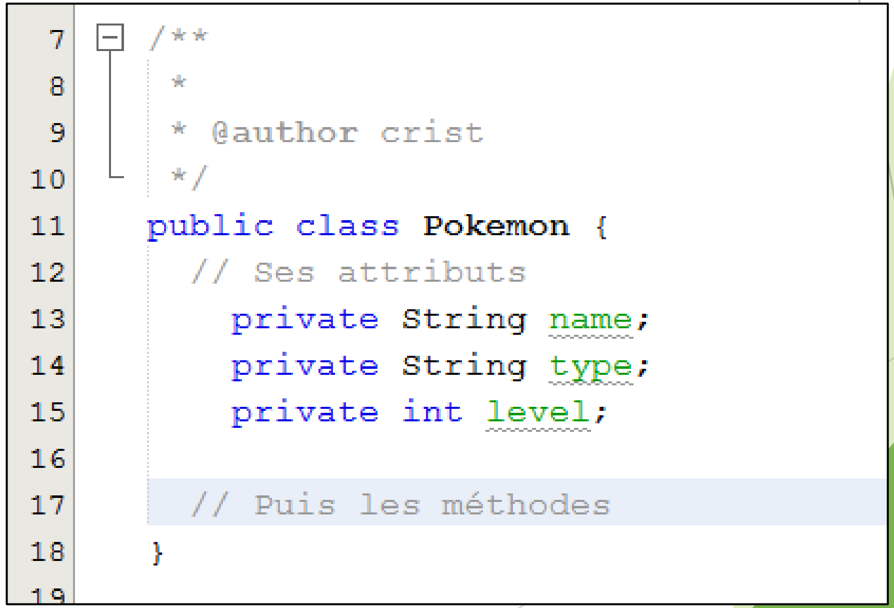
Basic Java syntax

Classes and objects (reminder)

- Class: an abstract representation (or model) of a concept
  - Examples: "Student", "Animal", "Computer", "Pokemon"...
  - Contains attributes and methods



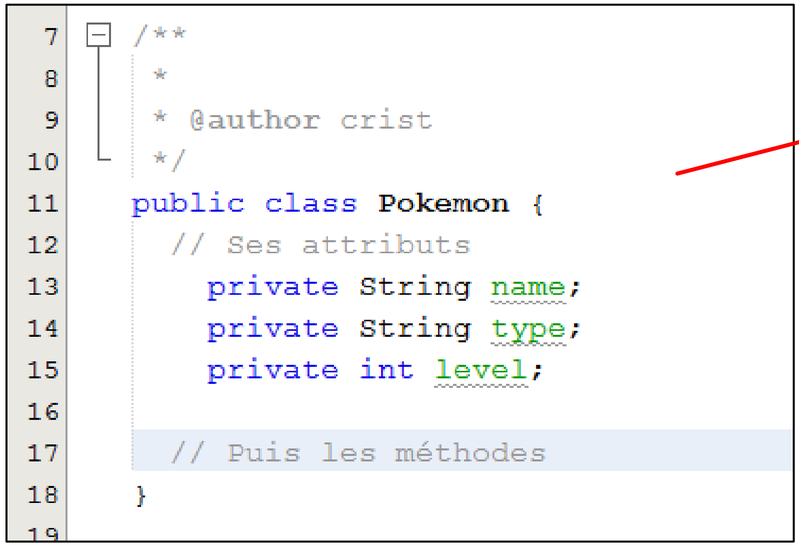
Variables that characterize the class



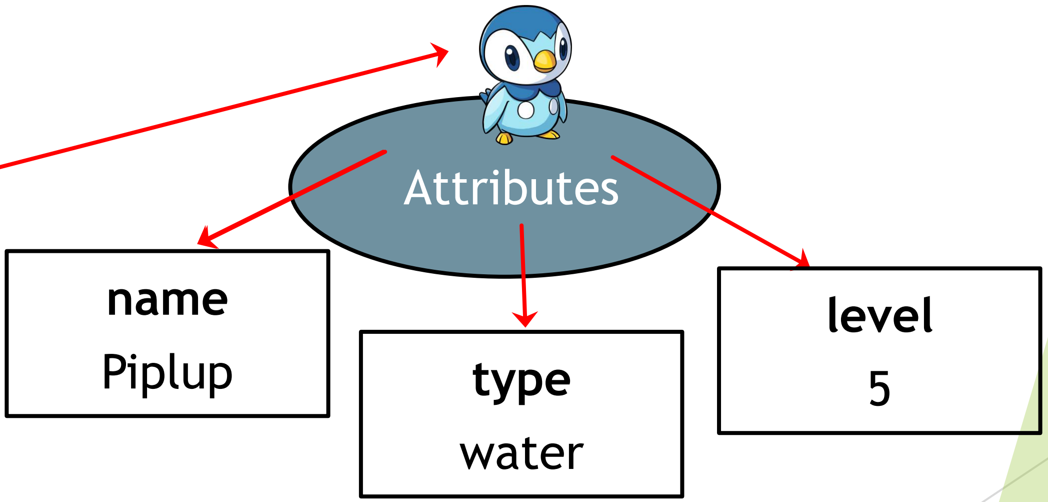
```
/**
 *
 * @author crist
 */
public class Pokemon {
  // Ses attributs
    private String name;
    private String type;
    private int level;

  // Puis les méthodes
}
```

Classes and objects (reminder)

- Class: an abstract representation (or model) of a concept
- In Java, each object instantiates the class that defines it
  - Each object is unique and must be customized

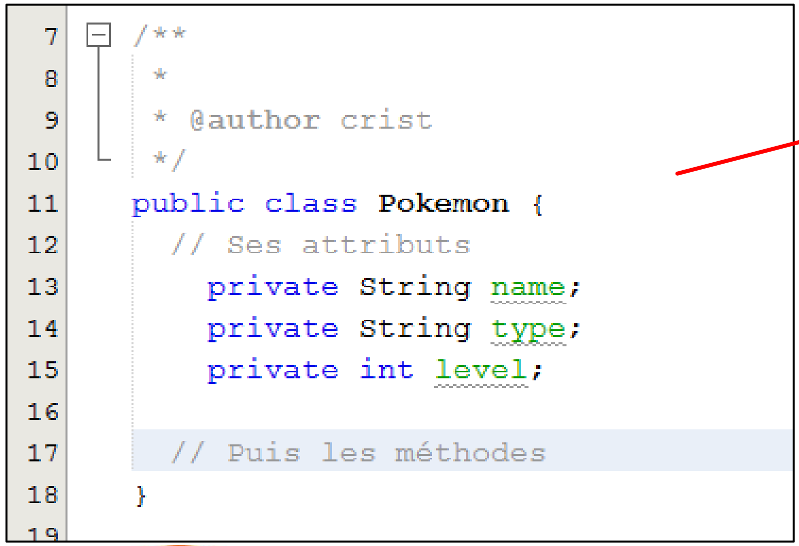


```
/**
 *
 * @author crist
 */
public class Pokemon {
  // Ses attributs
    private String name;
    private String type;
    private int level;

  // Puis les méthodes
}
```

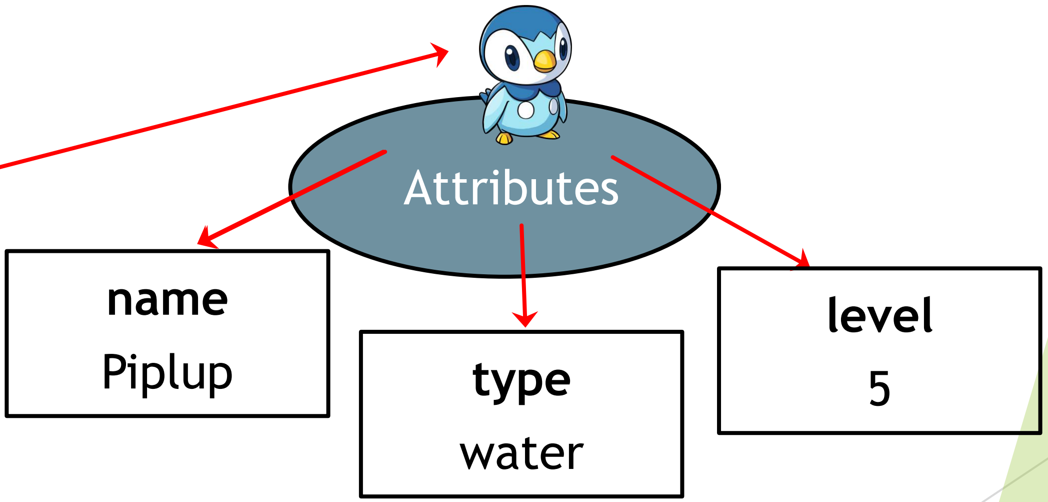


Classes and objects (reminder)

- Class: an abstract representation (or model) of a concept
- In Java, each object instantiates the class that defines it
  - Each object is unique and must be customized



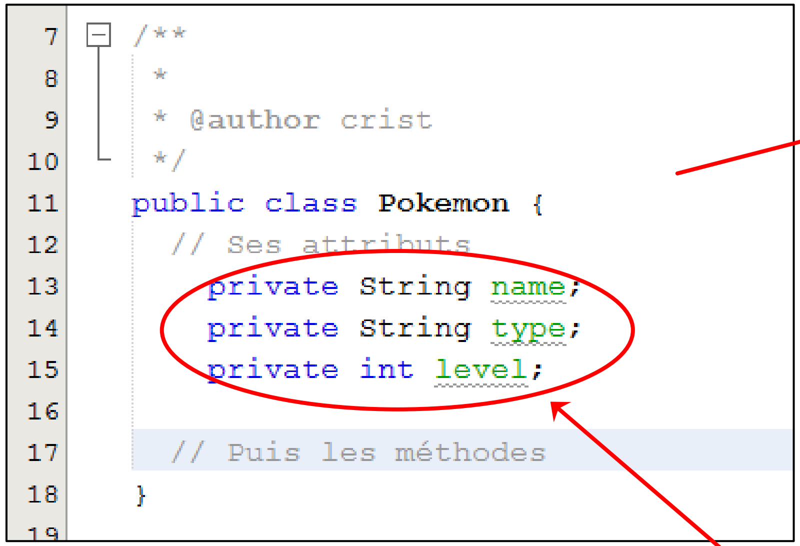
```java
/**
 *
 * @author crist
 */
public class Pokemon {
  // Ses attributs
    private String name;
    private String type;
    private int level;

  // Puis les méthodes
}
```



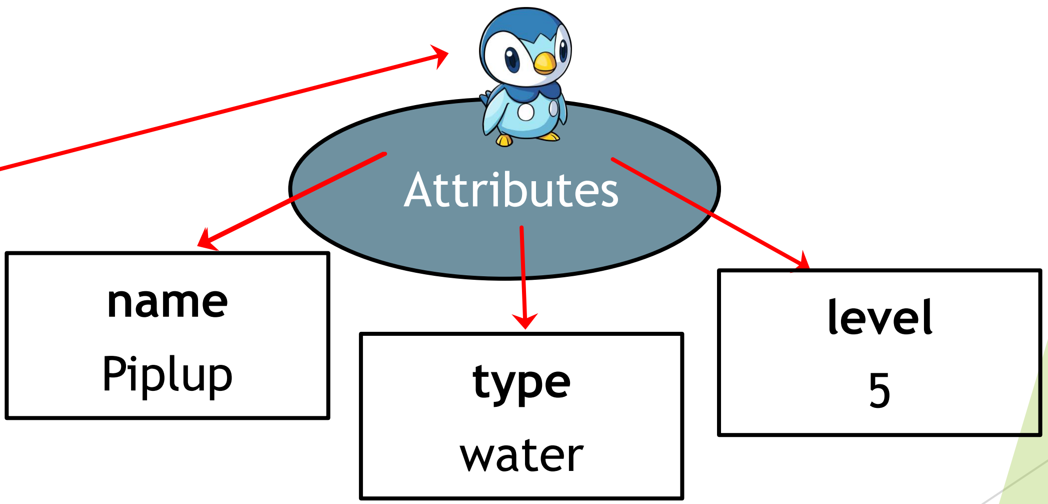
GP1 (Convention): class starts with capital letter, object starts with lowercase
ex: Pokemon vs. a pokemon

Classes and objects (reminder)

- Class: an abstract representation (or model) of a concept
- In Java, each object instantiates the class that defines it
    - Each object is unique and must be customized



```java
/**
 *
 * @author crist
 */
public class Pokemon {
    // Ses attributs
    private String name;
    private String type;
    private int level;

    // Puis les méthodes
}
```



Let's have a look at variables in Java!

Java variables: a howto

- Four steps in handling variables in Java:
  1. Declaring variables: visibility, type, name are stated

```
private String name; private Pokemon piplup;
```

  2. Instantiation: create an object (special method: constructor)

```
Pokemon piplup;  piplup = new Pokemon("Piplup", "Eau", 5);
```

  3. Assignment (initialisation): a first value is assigned to a variable

```
name = "Piplup"; age = 7;
```

  4. Modification/ré-assignement : cette valeur peut ensuite être modifiée

```
name = "Rowlet"; piplup = new Pokemon("Rowlet", "Herbe", 10);
```

Simultaneous declaration + instantiation:

```
Pokemon piplup = new Pokemon("Piplup", "eau", 5)
```

Variable types in Java

- Primitive types (8 in total!) :
  - start with lowercase letters

    `byte, short, int, long` – 8-, 16-, 32-, 64-bit long integers

    `float, double` – decimal numbers, written with a dot: 3.4, 1.7, ...

    `char` – 1 character, written between apostrophes: 'c', 'd', ...

    `boolean` – true/false

- Non-primitive types (Java classes):
  - `String` – character strings, written between inverted commas: "Piplup"
  - `Arrays`: a data structure
  - All other objects

Three types of variables

- **Case 1: class attributes (ex: name is an attribute of Pokemon)**
  - Declared at beginning of the class description (usually not instantiated)
  - Each attribute has a visibility: `public, private, protected, ...`

    ```
    private String name;     private int level;
    ```
  - Personalisable by each instance (each object)

    ex : each pokemon has a name, each has a level
  - **Case 2: special static attributes**

Three types of variables

- **Case 1: class attributes (ex: name is an attribute of Pokemon)**
  - Declared at beginning of the class description (usually not instantiated)
  - Each attribute has a visibility: `public, private, protected, ...`
  - Personalisable by each instance (each object)
  - Case 2: special static attributes

- **Case 3: other variables (appearing in and local to methods)**
  - Do not exist outside the environment for which they are defined
  - Declared before/upon first use
  - Using undeclared variables triggers an error of compilation

Intermezzo : compilation error

What is a compilation error ?

- Two types of errors in Java code : compilation and execution errors

- Compilation errors: code that is syntactically wrong
  - Like spelling or grammatical errors in French/English languages
  - The IDE detects those errors and signals it to the user

- Execution errors: code that is wrong for some particular exécution
  - Sentences that do not make sense in a text
  - The IDE cannot detect them, and they can crash the code
  - Can be treated by using exceptions

Errors: examples

- Compilation errors:
  - Using variables without declaring them
  - Bad use of code syntax, semicolons, etc.
  - Incorrect references to variables, etc.
  - …
- Execution errors:
  - Reading from or writing to a non-existent file
  - Referencing beyond the size of a data structure (like an array)
  - …

End of intermezzo

Basic Java instructions for variables

- **Assignment:**

```java
String pokemonName = "Piplup";    int level = 5;
```

; at end of line

- **Printing** a primitive variable:

```java
System.out.println(<variableName>);
```

Exceptionally usable for String variables
Later: how to use this for other objects

- **Testing equality** (primitive types): returns a boolean

```java
int a=2; int b=3; boolean equality = (a == b);
```

== : Equality test
= : assignment

Caution : non-primitive types do not work like primitive types !
For Strings: `String a = "un string";`
`String b = "un string";`
`(a ~~==~~ b);`
Comparing objects: use `a.equals(b)` !

Operations using variables

- Addition and subtraction :
  - numeric types: + is addition, - is subtraction,
  - boolean type: + and - do not apply
  - String : + indicates the concatenation of strings

```
System.out.println("Pip" + "lup");
```

```
>> Piplup
```

Caution : we do not use + on chars !

- Multiplication and division (* and /) : only numeric types
  - The result of dividing two integers is an integer by default. Java rounds the result automatically: 7/2 = 3
  - Obtain a correct result cast the type to a more suitable one

```
double result = (double) 7/2;
```

Variables and logic

- Boolean variables can be used with logical operators:
  - Negation: true → false and false → true;
    - Syntax : `!<variable>` or `!(<value>)` or `!=`
    - `!(a == b)` is the same as `(a != b)`

```
boolean isEqual;
isEqual = !(2==3);
System.out.println(isEqual);   >> true
System.out.println(5 == 6);    >> false
```

  - Logical OR: true/false OR true → true; false OR false → false
    - Syntax : `<boolean1> || <boolean2>`
    - Can apply to variables or expressions

```
boolean isEqual = (2!=3) || (5 == 6);
System.out.println(isEqual);        >> true
```

  - Logical AND: true/false AND false → false; true AND true → true
    - Syntaxe : `<boolean1> && <boolean2>`

```
boolean isEqual = (2!=3) && (5 == 6);
System.out.println(isEqual);        >> false
```

More advanced Java syntax

Strings

- String is a Java class, defining a type – hence the capital letter
- Strings are a special type, as they can be handled:
  - Similarly to primitive variables:

```
String pokemonName;
pokemonName = "Piplup";
```

  - As complex objects :

```
String pokemonName;
pokemonName = new String("Piplup");
```

GP2 : We will typically use the first of these methods…
… but we will remember that String is not a primitive type!

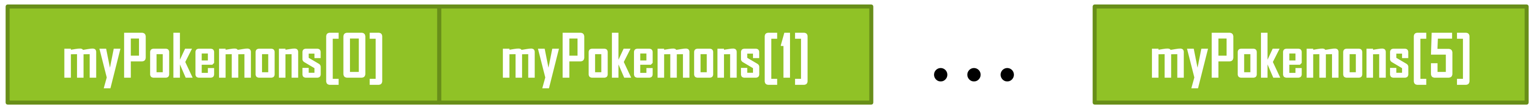
Arrays

- An array is an object which represents a collection of other objects
  - One main attribut: its length (# of objects contained)
- Use :
  1. Declaring an array : `<type>[] <name>`

```
double[] grades; Pokemon[] myPokemons;
```

  2. Instatiation: compulsory (exception on next page)
     - Defines length: `<name>=new <type>[<length>]`
     - Arrays are indexed, from 0 to `(length - 1)` :

```
myPokemons = new Pokemon[6]
```



Arrays

- An array is an object which represents a collection of other objects
  - One main attribut: its length (# of objects contained)
- Use :

1. Declaring an array : `<type>[] <name>`
2. Instatiation: compulsory (exception on next page)
   - Defines length: `<name>=new <type>[<length>]`
   - Arrays are indexed, from 0 to (length - 1) :
3. Assignment: three ways:
   - Instantiation + assignment:
   - Implicit length by assig
   - Element by element:

```
double[] grades=new double[3];
  grades={12.0, 16.5, 13.0};
```

```
6.5, 13.0};
{18,25}
```

```
double[] grades = new double[3];
grades[0]=12.0;
grades[1]=16.5;
grades[2]=13.0;
```

Operations with arrays

- Array elements "borrow" all operations belonging to their types:
  - Ex.: the elements of a String[] can use any operation native to Strings
    - comparison: `<string1>.equals(<string2>)`
    - + allows the concatenation of Strings
    - = is used for assignment -- remember also to use the inverted commas " "
- Arrays can also be manipulated on their own:
  - However, such operations should be handled with care!

```
double[] myGrades = {12, 10, 15.6};
double[] yourGrades = myGrades;
myGrades[2] = 13;
System.out.println(yourGrades[2]);
```

Initialise myGrades
set yourGrades = myGrades

Modify myGrades[2]

```
>> 13
```

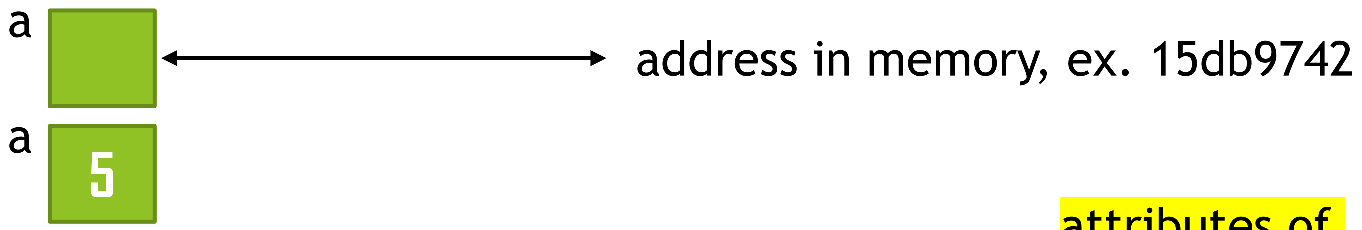
Why ??

Variables stored in memory

- Every variable and every object is stored in memory:

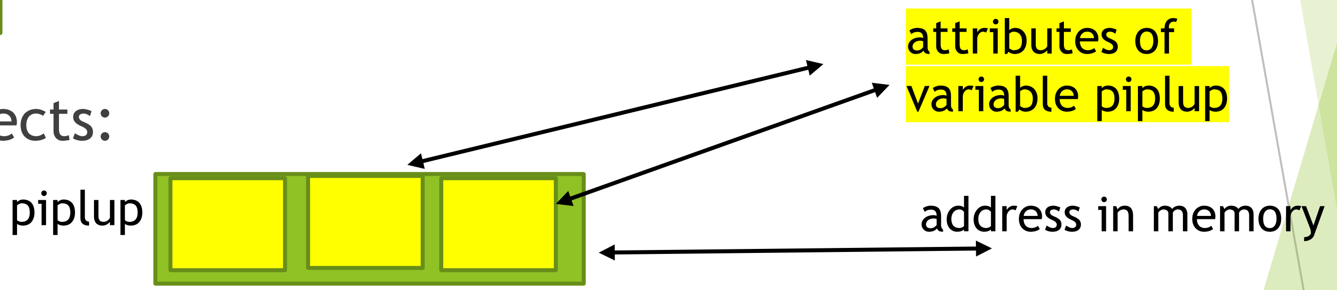
```
int a;
```

```
a = 5;
```



- This also holds for objects:

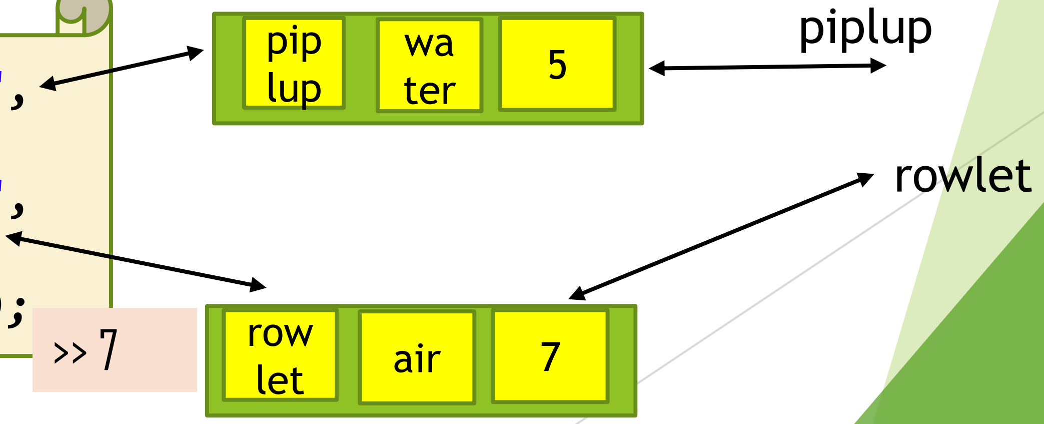
```
Pokemon piplup;
```



- Assignment:

```
Pokemon piplup = new Pokemon("Piplup",
"WATER", 5);
Pokemon rowlet = new Pokemon("Rowlet",
"AIR", 7);
System.out.println(rowlet.getLevel());
```

```
>> 7
```

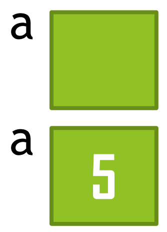


Variables stored in memory

- Every variable and every object is stored in memory:

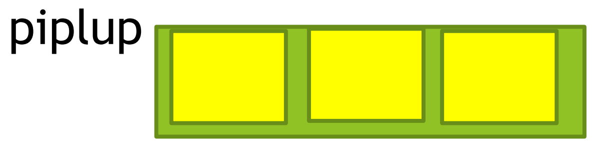
```
int a;
```

```
a = 5;
```



- This also holds for objects:

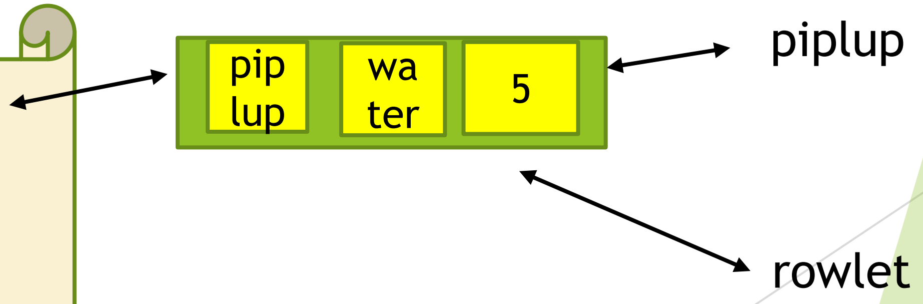
```
Pokemon piplup;
```



- Assignment:

```
Pokemon piplup = new Pokemon("Piplup",
"WATER", 5);
Pokemon rowlet = piplup;
piplup.setLevel(7);
System.out.println(rowlet.getLevel());
```

```
>> 7
```



These variables share an addresss
Modifying one changes the other

Conditional execution (if-then-else)

- Syntax:

```
if (<test>) {
    // instructions separated by ";"
}
else {
    // instructions separated by ";"
}
```

in-code comment
(compiler disregards text on same line after //)

```
if (2==3){
    System.out.println("Blue pill.");
}
else {
    System.out.println("Red pill.");
}
```

- One instruction => curly brackets {} are not compulsory

GP3 : properly indent your code (indentation is 2 to 4 characters)
GP4 : use the curly brackets! (we always will)

While loops

- Syntax

```
while (condition) {
    // instructions separated by ";"
}
```

stand-alone method
(in a class)

return type : int
visibility : public

```
// compute 1+2+...+100

public int sum1to100(){
    int result = 0;
    int i = 1; //iterator
    while (i <= 100){
        result +=i;
        i++;
    }
    return result;
}
```

iterator local to method

- Remember to increment the iterator

For loop

- Syntax:

```
for (<start condition>; <stop condition>; <incrementation>) {
        // instructions separated by ";"
}
```

```
// compute 1+2+...+100

public int sum1to100(){
    int result = 0;
    for (int i=1; i<=100; i++){
        result +=i;
    }
    return result;
}
```

- Remember to declare the iterator !

Methods in Java

Why methods?

- Java methods allow us to:
  - instantiate classes (special method called a constructor)
  - initialize or modify the values of an attribute
  - do a computation on the attributes in a class
  - obtain a result, such as printing on the screen
  - ...
- All methods in Java are included in classes
  - Most methods in a class are run "by" (or for) given instances of that class
  - An exception is using a static method

Attributes and methods

- Here's a Pokemon class:
  - Attributes go at the top
  - Method 1: Pokemon (constructor)

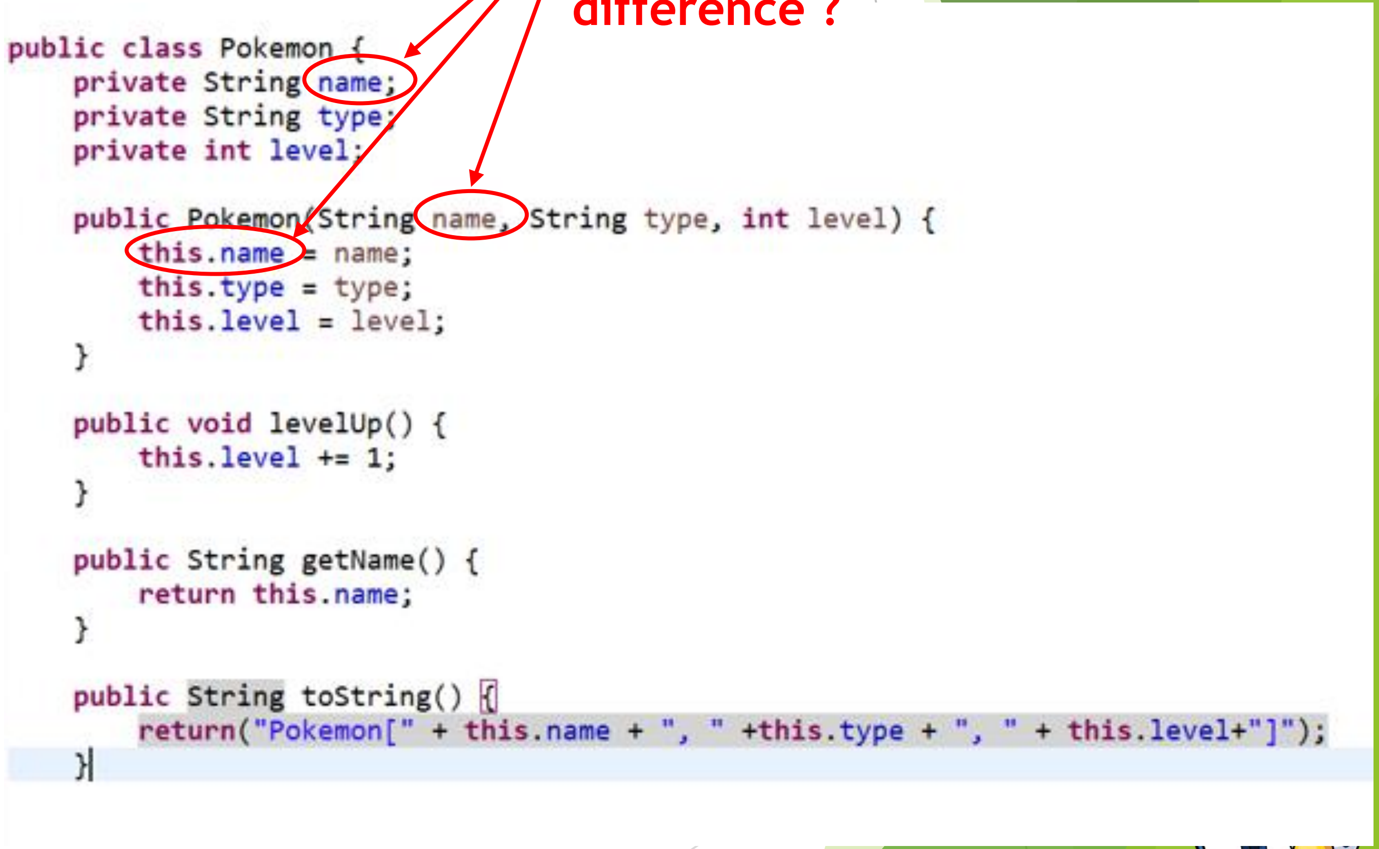
    Allows to instantiate pokemons
  - Method 2: levelUp

    Modifies an attribute
  - Method 3: getName (a getter)

    Retrieves attribute (level)
  - Method 4: toString

    Special role we will see later

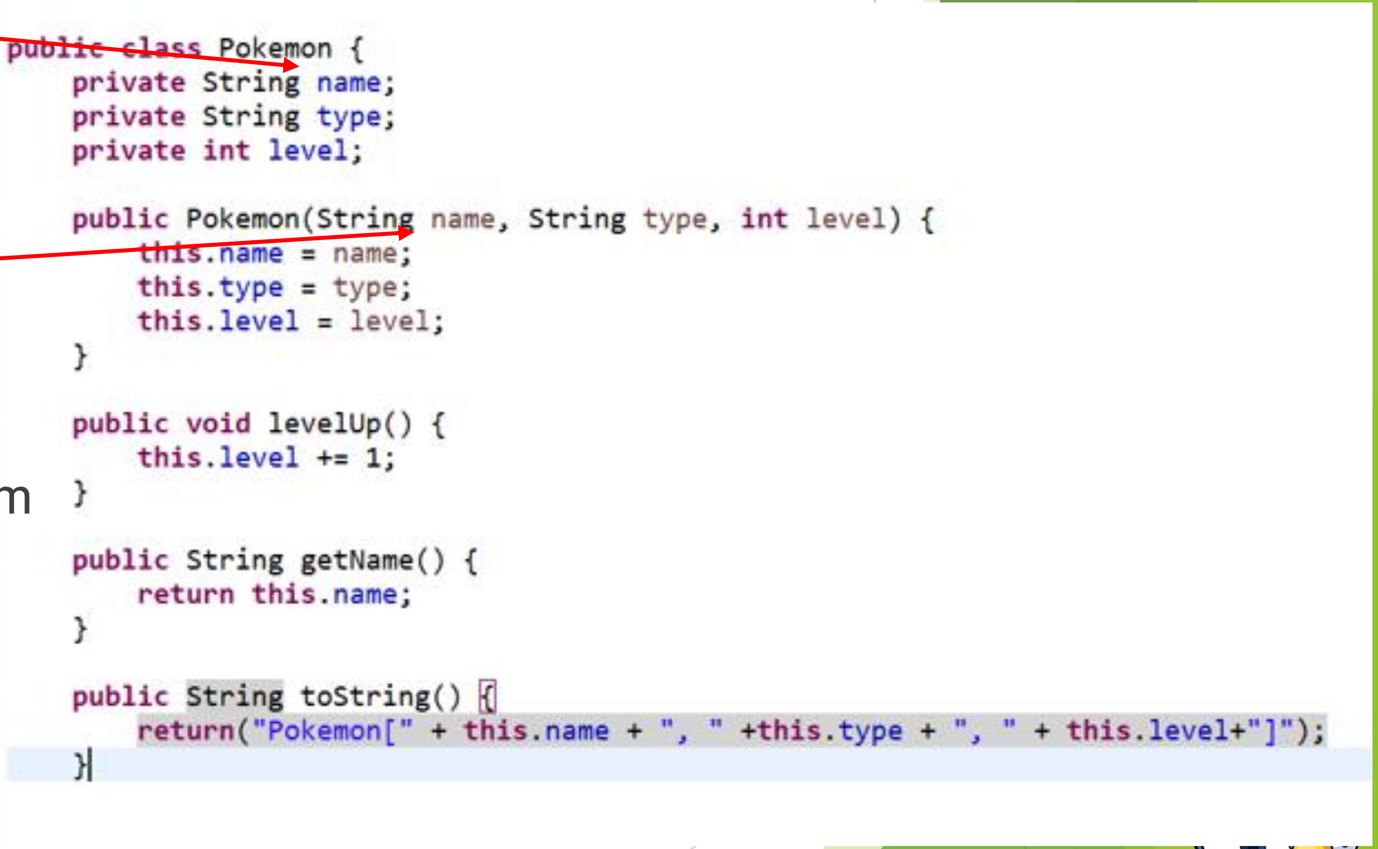
What's the difference ?



```java
public class Pokemon {
    private String name;
    private String type;
    private int level;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
    }

    public void levelUp() {
        this.level += 1;
    }

    public String getName() {
        return this.name;
    }

    public String toString() {
        return("Pokemon[" + this.name + ", " +this.type + ", " + this.level+"]");
    }
```

Variables, attributes, parameters

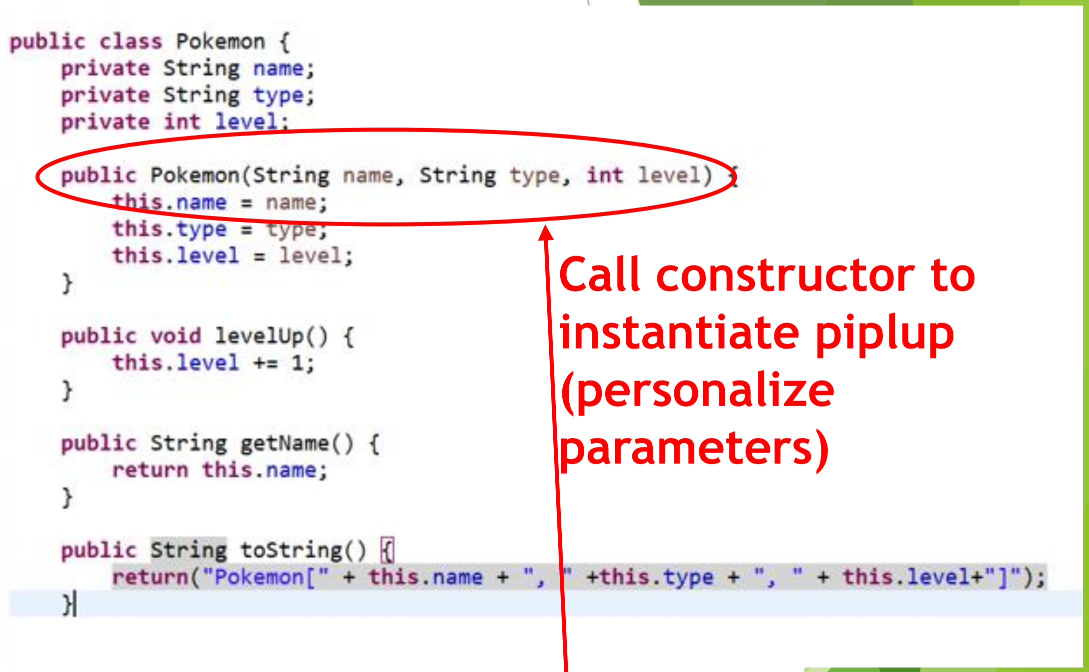
- Attributes:
  - Variables that characterize a class
  - Declared at the top of the class
  - Instantiated in constructor
- Parameters:
  - Variables input to methods
  - Symbolic at method declaration
  - Each call to method personalises them
- Other variables:
  - Local to methods
  - Used for storage, iteration



```
public class Pokemon {
    private String name;
    private String type;
    private int level;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
    }

    public void levelUp() {
        this.level += 1;
    }

    public String getName() {
        return this.name;
    }

    public String toString() {
        return("Pokemon[" + this.name + ", " +this.type + ", " + this.level+"]");
    }
```

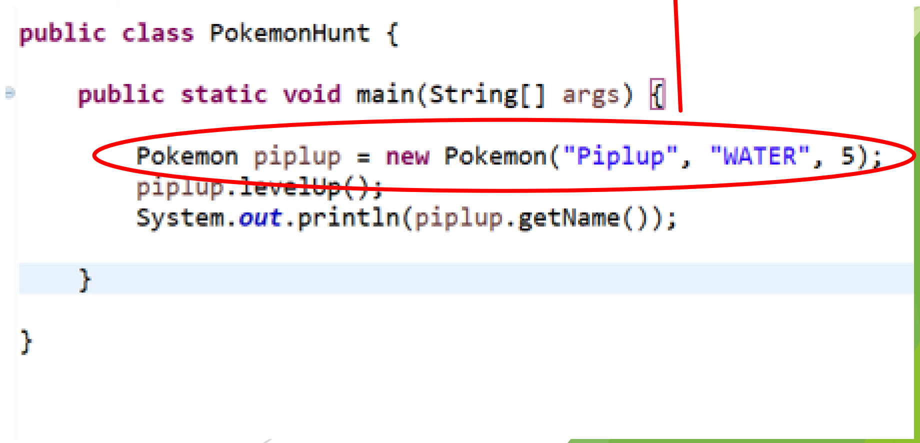
Attributes, methods, and parameters

- Methods in Java appear in two places:
  - When they are defined (inside their class)
  - When they are used (inside our outside class)
- Defining (describing) methods:
  - Optionally use a number of parameters
  - Tell us output type
  - For concrete methods: write out the code
- Using methods:
  - "Personalize" parameters to what we want
  - Call method for object
  - public methods can be called outside class; private methods cannot



```java
public class Pokemon {
    private String name;
    private String type;
    private int level;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
    }

    public void levelUp() {
        this.level += 1;
    }

    public String getName() {
        return this.name;
    }

    public String toString() {
        return("Pokemon[" + this.name + ", " +this.type + ", " + this.level+"]");
    }
```

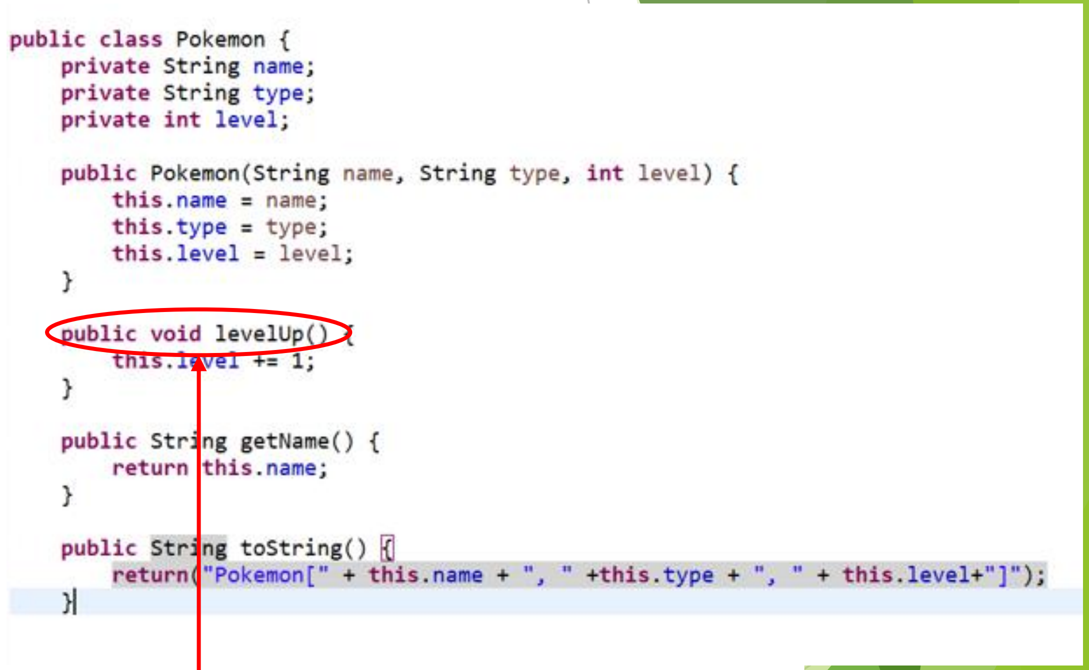
Call constructor to instantiate piplup (personalize parameters)



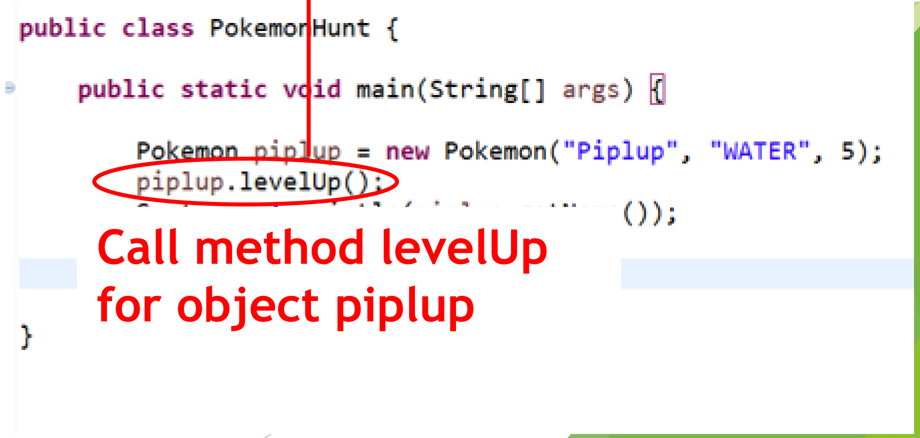
```java
public class PokemonHunt {

    public static void main(String[] args) {

        Pokemon piplup = new Pokemon("Piplup", "WATER", 5);
        piplup.levelUp();
        System.out.println(piplup.getName());

    }

}
```

Attributes, methods, and parameters

- Methods in Java appear in two places:
  - When they are defined (inside their class)
  - When they are used (inside our outside class)
- Defining (describing) methods:
  - Optionally use a number of parameters
  - Tell us output type
  - For concrete methods: write out the code
- Using methods:
  - "Personalize" parameters to what we want
  - Call method for object
  - public methods can be called outside class; private methods cannot



```java
public class Pokemon {
    private String name;
    private String type;
    private int level;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
    }

    public void levelUp() {
        this.level += 1;
    }

    public String getName() {
        return this.name;
    }

    public String toString() {
        return("Pokemon[" + this.name + ", " +this.type + ", " + this.level+"]");
    }
```



```java
public class PokemonHunt {

    public static void main(String[] args) {

        Pokemon piplup = new Pokemon("Piplup", "WATER", 5);
        piplup.levelUp();
        [illegible]());

}
```

Call method levelUp for object piplup

Attributes and methods

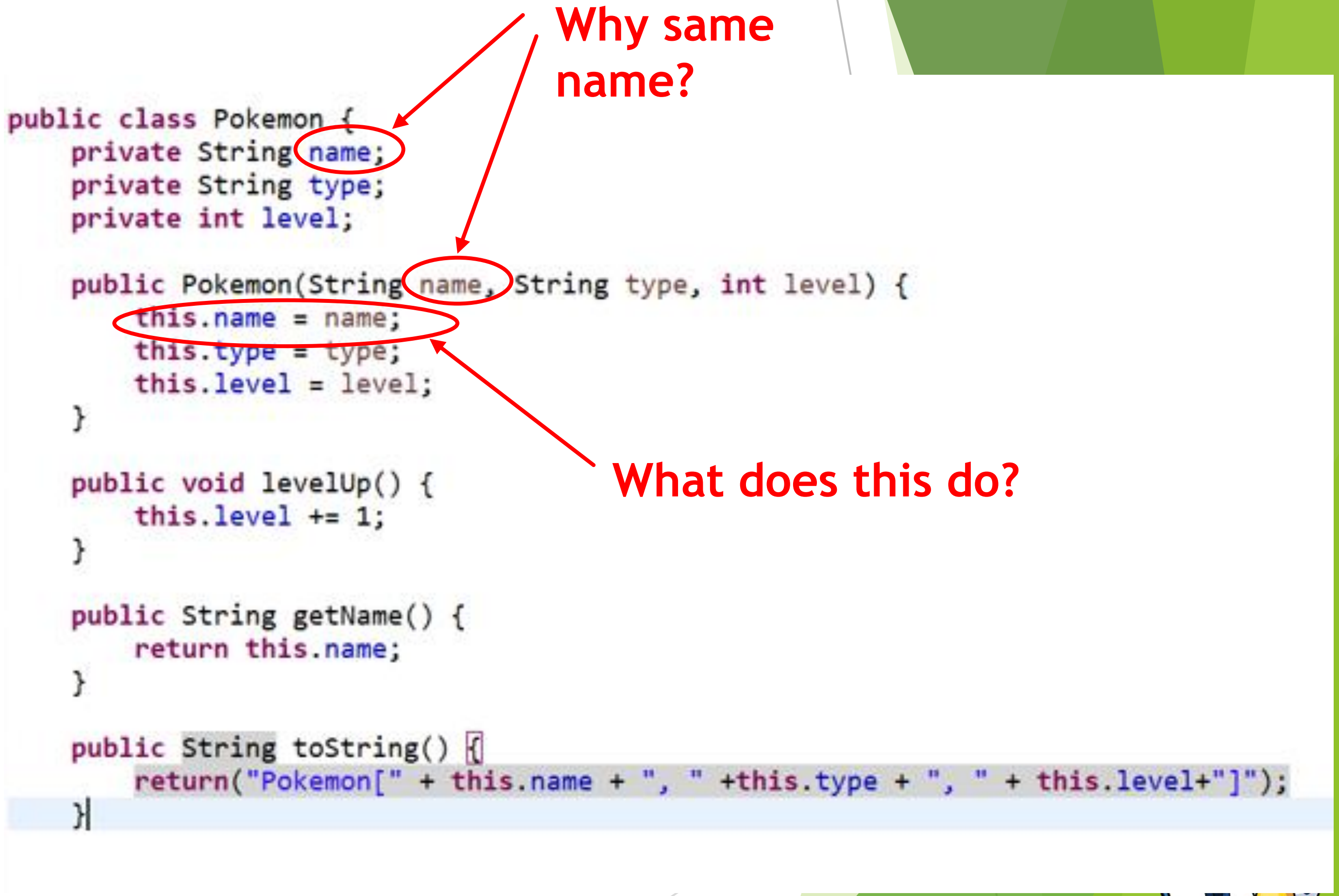
- Here's a Pokemon class:
  - Attributes go at the top
  - Method 1: Pokemon (constructor)

    Allows to instantiate pokemons
  - Method 2: levelUp

    Modifies an attribute
  - Method 3: getName (a getter)

    Retrieves attribute (level)
  - Method 4: toString

    Special role we will see later



```
public class Pokemon {
    private String name;
    private String type;
    private int level;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
    }

    public void levelUp() {
        this.level += 1;
    }

    public String getName() {
        return this.name;
    }

    public String toString() {
        return("Pokemon[" + this.name + ", " +this.type + ", " + this.level+"]");
    }
```

Variable references in Java

- **<u>Case 1: attribute (ex. class Pokemon)</u>**
  - Reference within class Pokemon: `this.<attributName>`

    Examples : `this.name`, `this.type`
  - Reference outside class: depends on visibility
    - Public: object piplup: `piplup.<attributName>`
    - Private: need to use special methods, like getters or setters
  - Special case: static attributes → Later!
- **<u>Case 2: not an attribute</u>**
  - Cannot be referenced outside of that method
  - Reference by name only

Examples

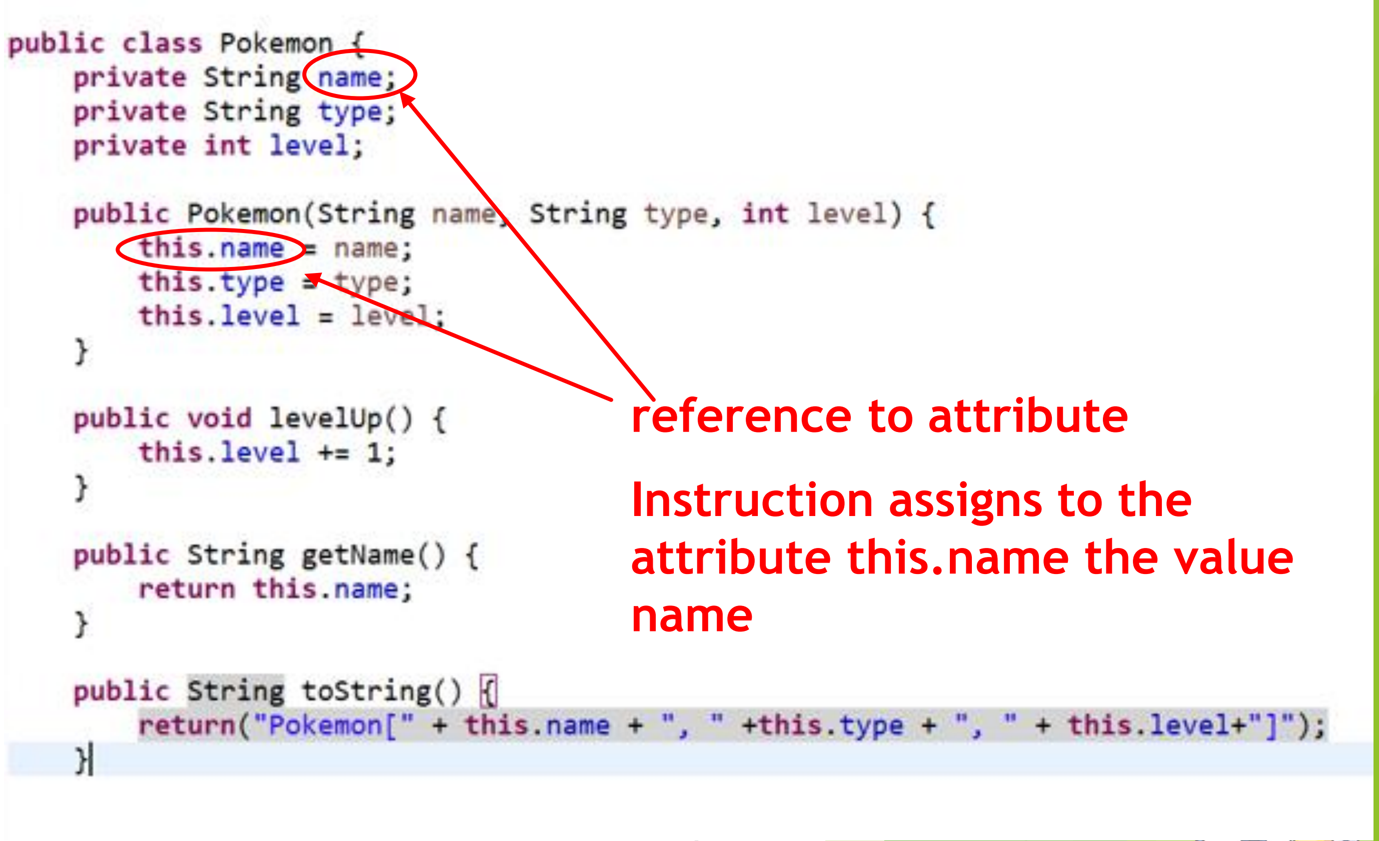
- Here's a Pokemon class:
  - Attributes go at the top
  - Method 1: Pokemon (constructor)

    Allows to instantiate pokemons
  - Method 2: levelUp

    Modifies an attribute
  - Method 3: getName (a getter)

    Retrieves attribute (level)
  - Method 4: toString

    Special role we will see later

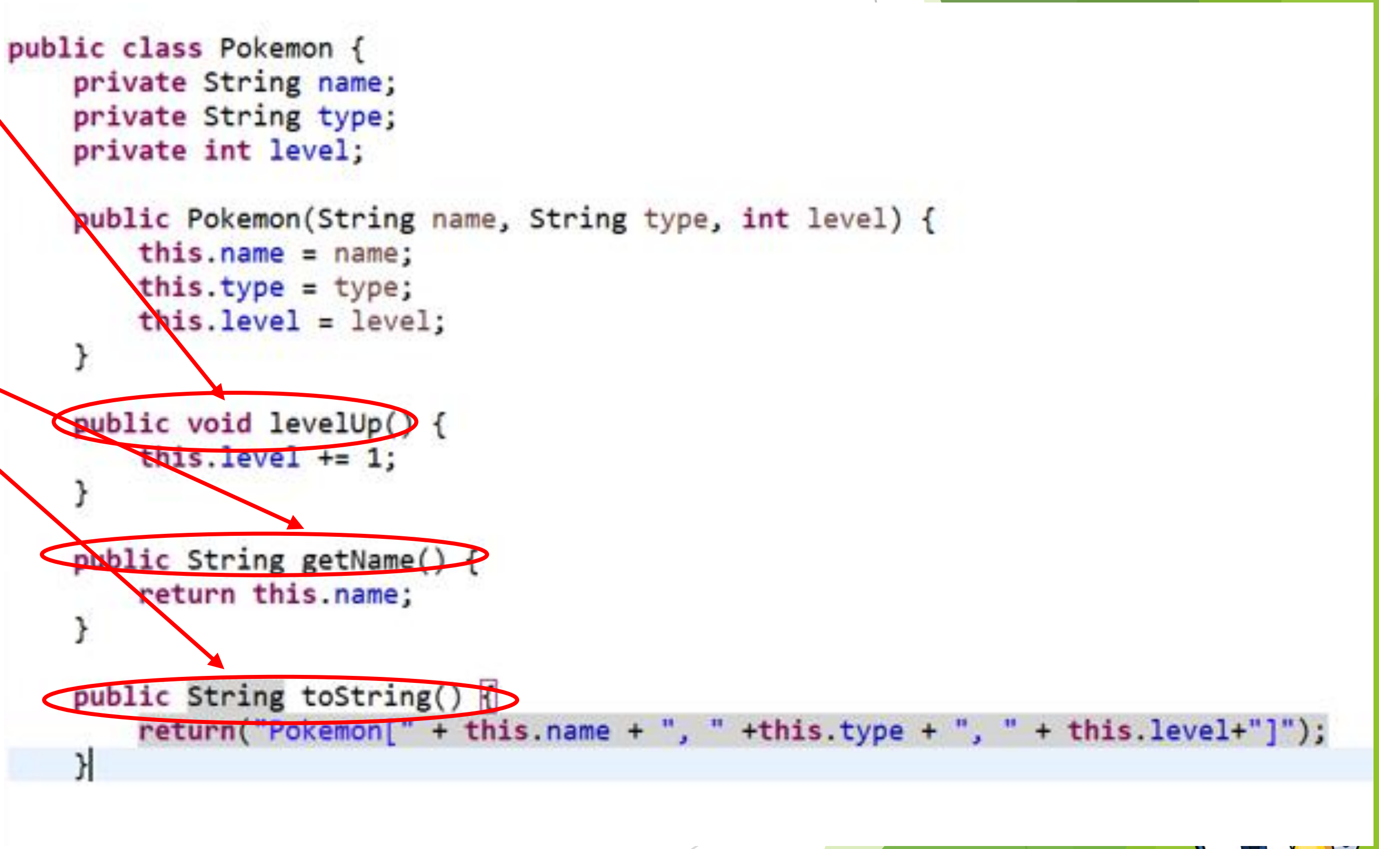


```
public class Pokemon {
    private String name;
    private String type;
    private int level;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
    }

    public void levelUp() {
        this.level += 1;
    }

    public String getName() {
        return this.name;
    }

    public String toString() {
        return("Pokemon[" + this.name + ", " +this.type + ", " + this.level+"]");
    }
```

reference to attribute
Instruction assigns to the attribute this.name the value name

Procedures and functions

- Procedure (output type void):
  - Modify an attribute
  - Assign an attribute for the first time
- Function (non-void output):
  - Requires a return of the declared type
  - The current branch of code will disregard instructions after return



```
public class Pokemon {
    private String name;
    private String type;
    private int level;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
    }

    public void levelUp() {
        this.level += 1;
    }

    public String getName() {
        return this.name;
    }

    public String toString() {
        return("Pokemon[" + this.name + ", " +this.type + ", " + this.level+"]");
    }
```

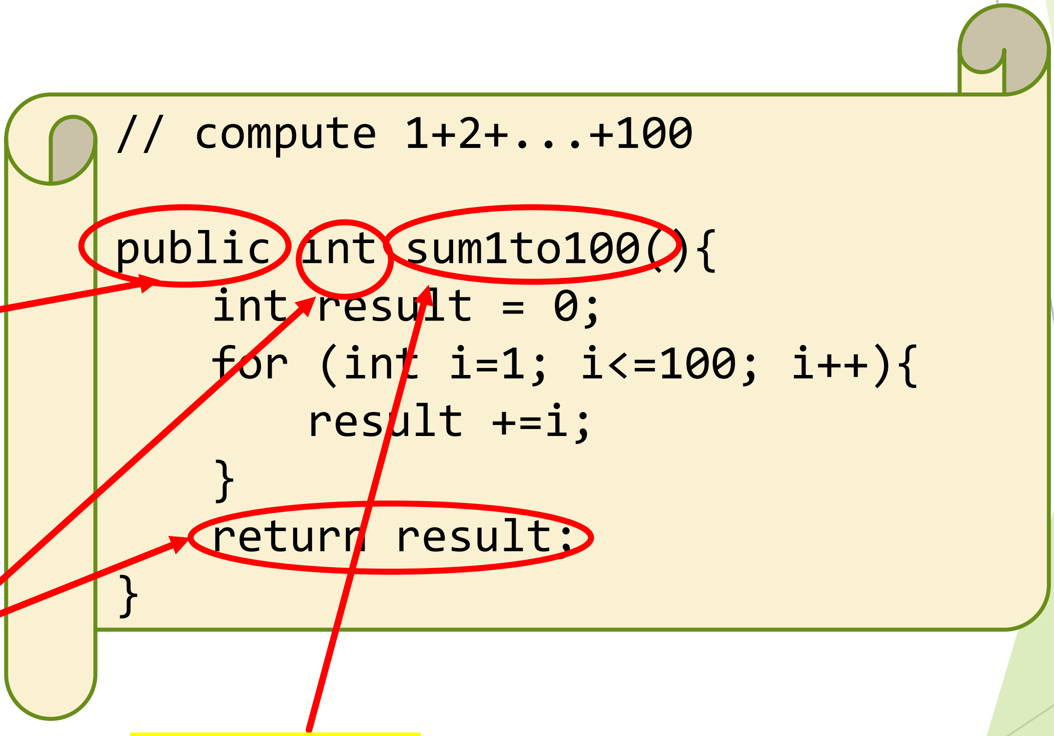
Methods and signatures

- Java methods are characterized by signatures, containing class and :
  - a return type (type of the variable to return) or void (no return)
  - the method's name
  - the types of the input variables (called the parameters)

- Syntax:

```
<visibility> <returnType> <name>(<typeP1> <nameP1>, <typeP2> <nameP2>,...) {
        // method contents
        // if method has non-void output type, it ends with a return statement
}
```

Example: compute 1+2+...+100



```
// compute 1+2+...+100

public int sum1to100(){
    int result = 0;
    for (int i=1; i<=100; i++){
        result +=i;
    }
    return result;
}
```

visibility: public method
(can be called from outside the class
where it is written)

The method returns an
integer value

Method name

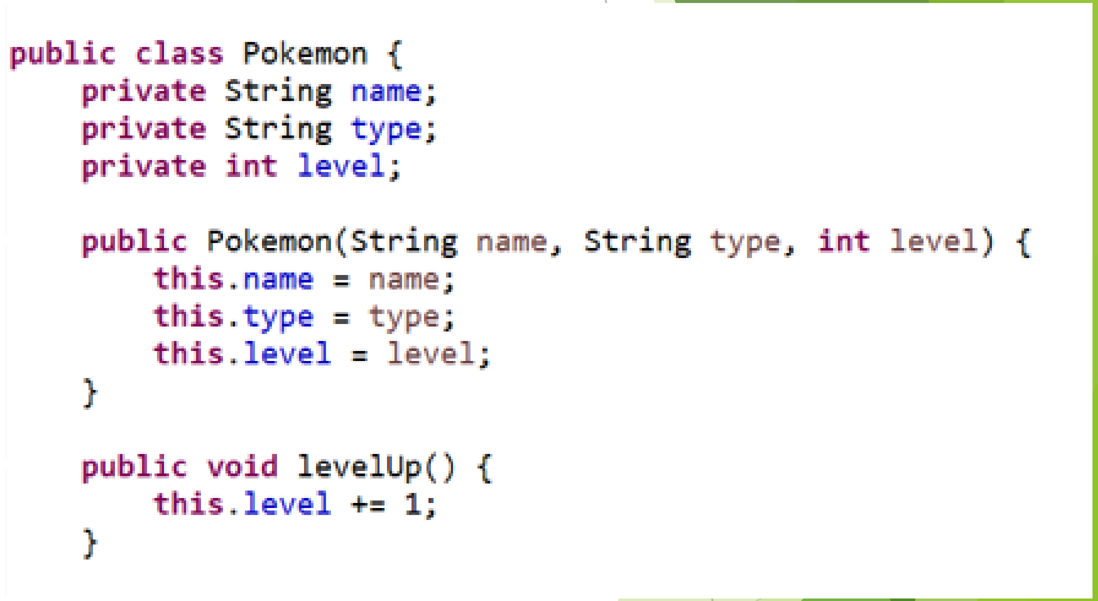
Special methods in Java

Special methods: Constructors

- Method names can be chosen at will

  GP5 : Keep them intuitive though!

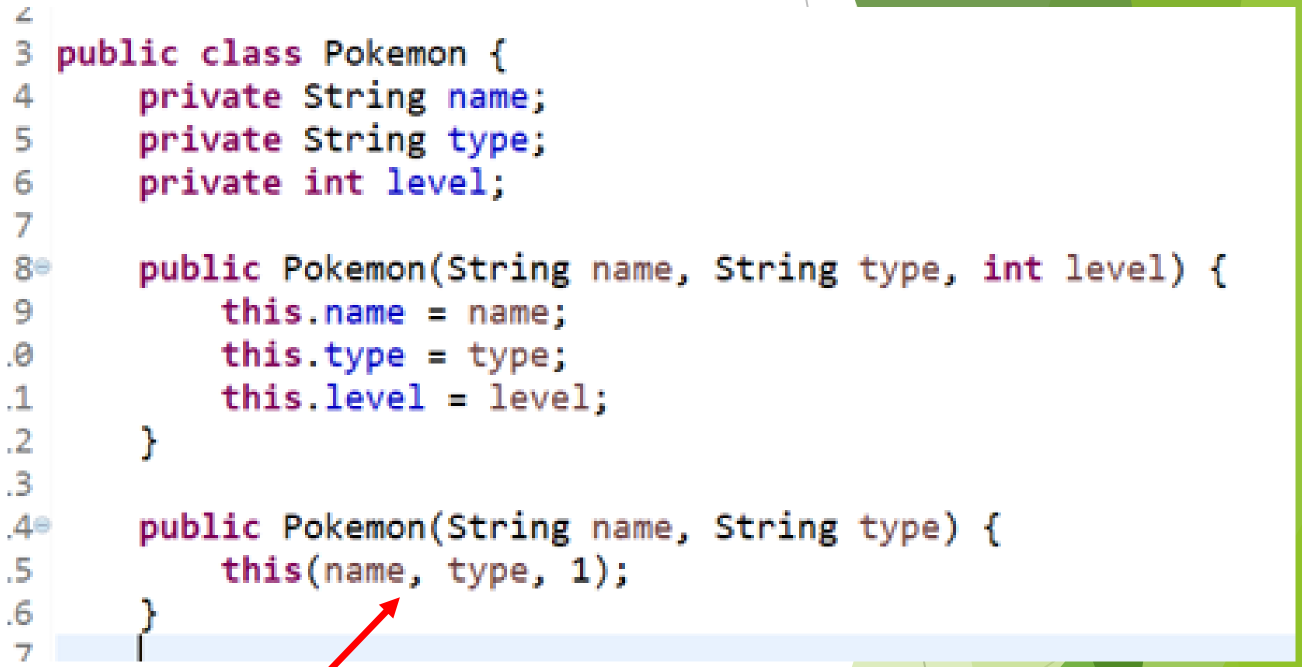
- Exception #1: constructors!
  - A special method that is used to instantiate objects
    - We usually initialize the class attributes within the constructor
    - Thus, objects personalize the class
  - Constructors are usually public
  - Constructors must be named after the class



```java
public class Pokemon {
    private String name;
    private String type;
    private int level;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
    }

    public void levelUp() {
        this.level += 1;
    }
```

Constructors: howto

- It is not compulsory to write constructors for each class
  - Java has a constructor by default
  - Signature <className>()
  - Constructors by default can be used to create objects but not to initialize their attributes
- Multiple constructors
  - All named after the class
  - But must have different signatures!
  - Typically, write the constructor with the most parameters, then call it in the other constructor(s)



```
public class Pokemon {
    private String name;
    private String type;
    private int level;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
    }

    public Pokemon(String name, String type) {
        this(name, type, 1);
    }
```

this : replaces Pokemon = constructor

uses the name/type from parameters but sets level to 1

Default constructors

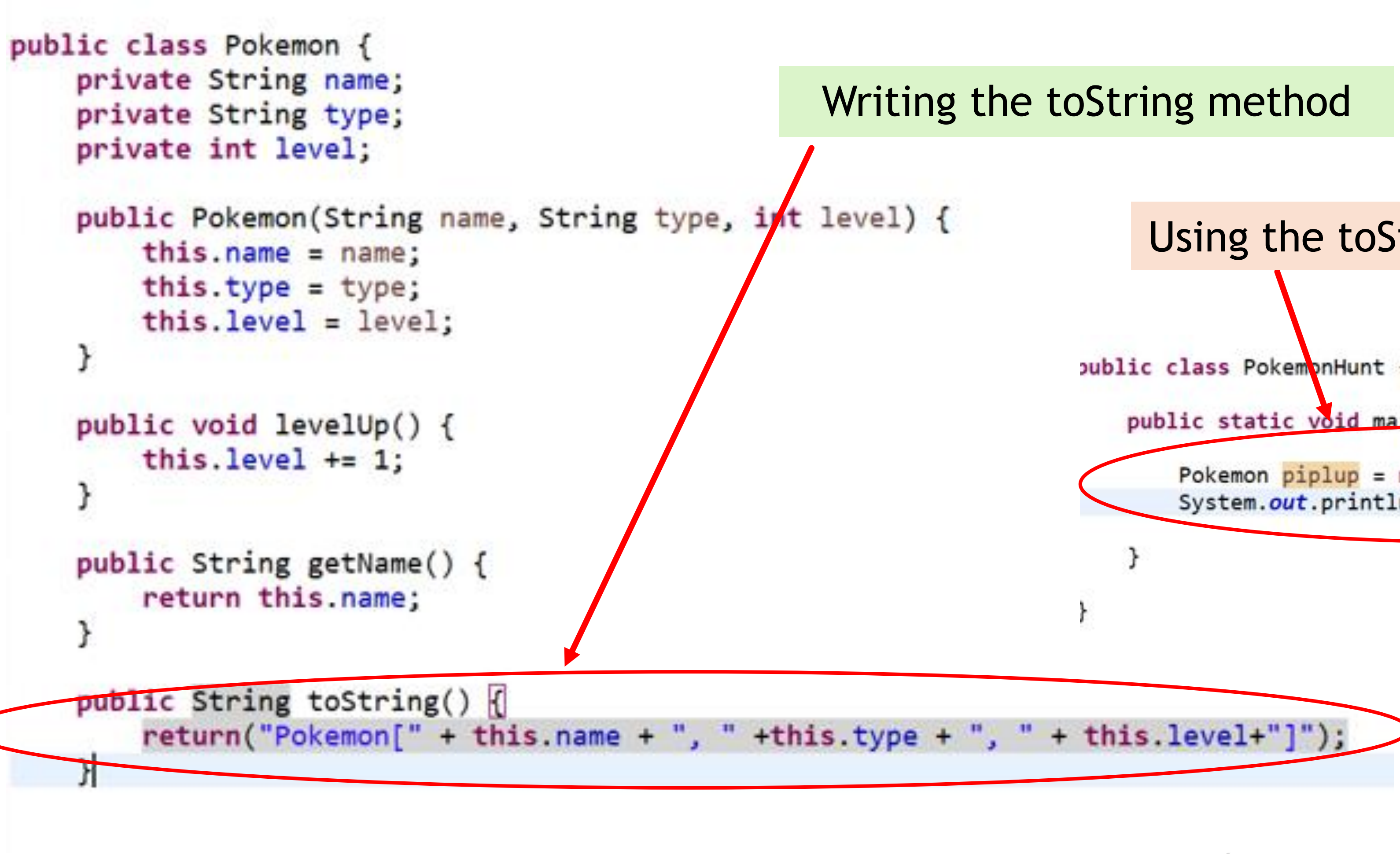
- Java.lang.Object is a basic class in Java
  - Which comes with a constructor
- All other classes in Java behave like Object's
  - We say they "inherit" from Java.lang.Object
- If a class does not have a constructor, it can fall back on Object's
  - Unfortunately this will not customize the objects
- However, as soon as the class gets its first constructor, it can no longer use the constructor by default

The String toString() method

- Printing a primitive or S[illegible] **Why?** ble: use `System.out.println`!
  - However, using System.out.println(piplup) will print a memory address
- To tell Java what you want to print for new class: use `String toString()`
- Writing `String toString()`: requires us to return a String
  - Typically, a concatenation of the attributes
  - Essentially "maps" each object to what we would like it to print as
- Calling a concrete String toString() method -- ex.: piplup.toString()
- Using a concrete String toString() method: System.out.println(piplup)

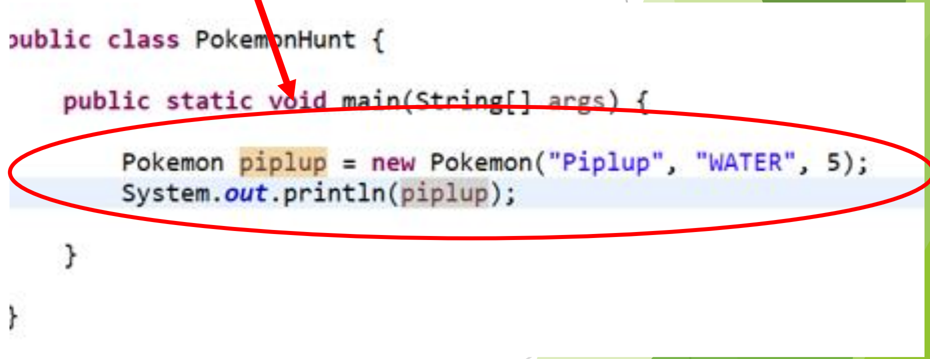
String toString() for Pokemon

Writing the toString method



```
public class Pokemon {
    private String name;
    private String type;
    private int level;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
    }

    public void levelUp() {
        this.level += 1;
    }

    public String getName() {
        return this.name;
    }

    public String toString() {
        return("Pokemon[" + this.name + ", " +this.type + ", " + this.level+"]");
    }
```

Using the toString method



```
ublic class PokemonHunt {

    public static void main(String[] args) {

        Pokemon piplup = new Pokemon("Piplup", "WATER", 5);
        System.out.println(piplup);

    }

}
```

Getters and setters

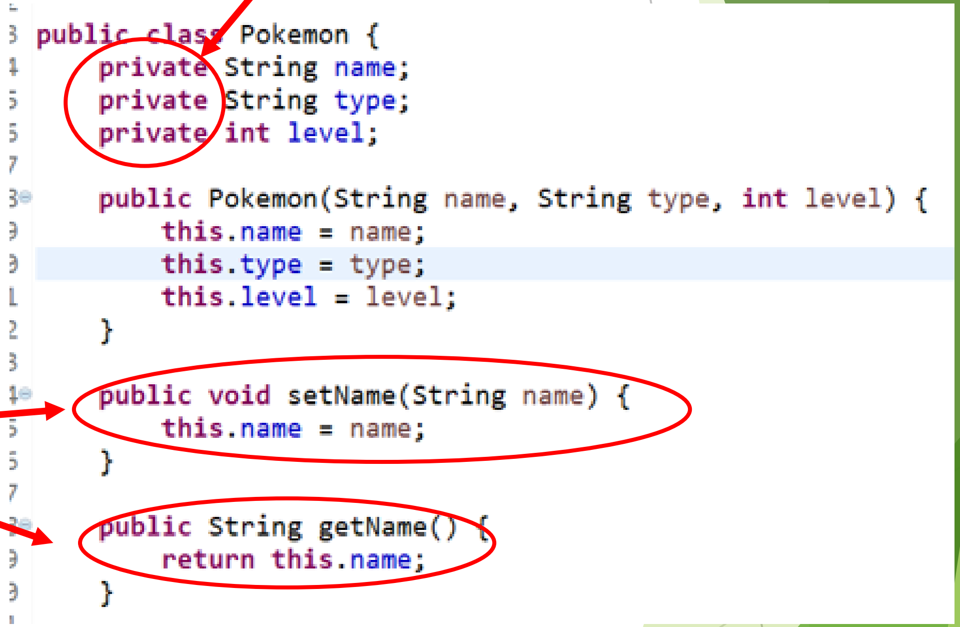
- Special methods that enable us to work with private attributes
  - Usually public visibility

- Getter:
  - retrives the attribute's current value

  ```
  <attributeType> get<attributeName>()
  ```

- Setter:
  - modifies the attribute's current value

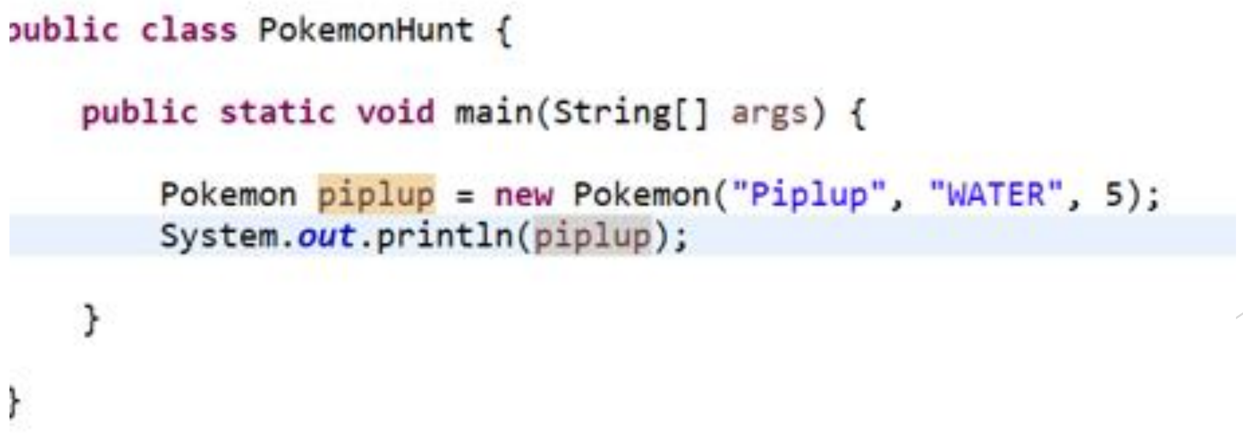
  ```
  void set<attributeName>(<attributeType> value)
  ```



```
public class Pokemon {
    private String name;
    private String type;
    private int level;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
    }

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

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

The main method

- The user's entry point into the program
- Included within a class (like all other methods in Java)
- Returns no output (void), takes in input a `String[]` array `args`
  - args can be used to parametrize the execution of the program
- This method is static (universal to all objects of this type)



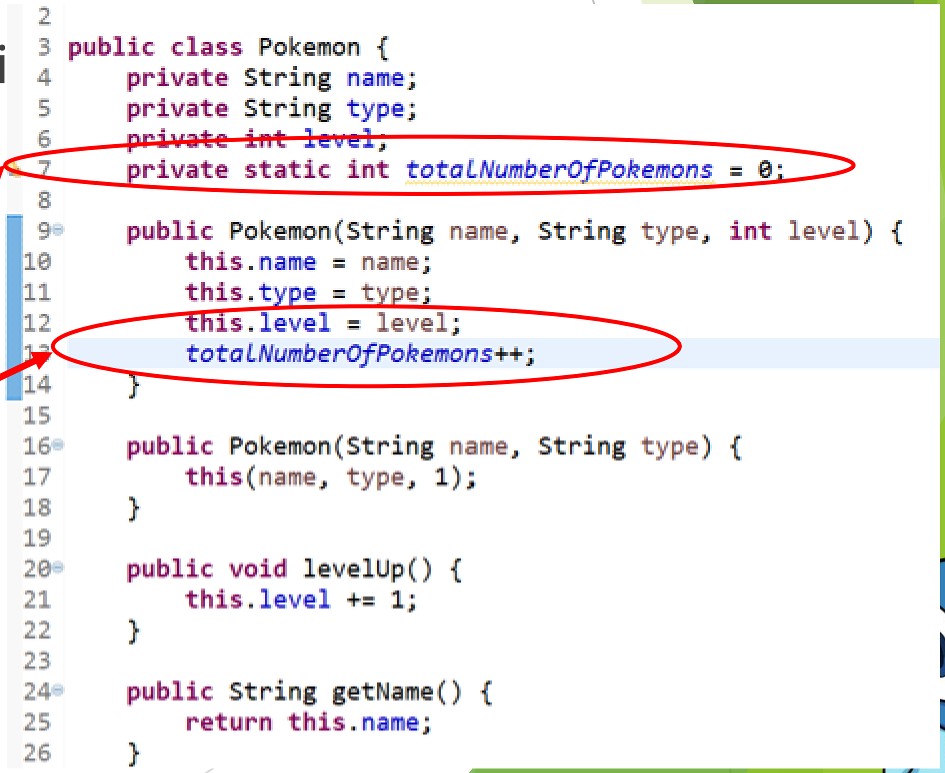
```
ublic class PokemonHunt {

    public static void main(String[] args) {

        Pokemon piplup = new Pokemon("Piplup", "WATER", 5);
        System.out.println(piplup);

    }

}
```

Static attributes and methods

- Attributes characterize a class
  - But each instance of that class has customized attributes
  - Changing the level of one pokemon does not typically affect another
- Static attributes are universal
  - Not custom to any instance of the class
  - ... but they apply to all instances
  - For instance, I could have a static counter of
    all pokemons ever created

Static attributes and methods

- Attributes characterize a class
  - But each instance of that class has customized attributes
  - Changing the level of one pokemon does not typi
- Static attributes are universal
  - Not custom to any instance of the class
  - ... but they apply to all instances
  - For instance, I could have a static counter of all pokemons ever created



```java

public class Pokemon {
    private String name;
    private String type;
    private int level;
    private static int totalNumberOfPokemons = 0;

    public Pokemon(String name, String type, int level) {
        this.name = name;
        this.type = type;
        this.level = level;
        totalNumberOfPokemons++;
    }

    public Pokemon(String name, String type) {
        this(name, type, 1);
    }

    public void levelUp() {
        this.level += 1;
    }

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

Accessing static attributes

- Usual attributes :
  - accessed for an instance of that class :
    - Directly (public attributes): piplup.name   if name is public
    - Indirectly (non-public attributes), using getters/setters: `piplup.getName()`
- Static attributes
  - can be accessed for an instance of that class: `piplup.totalNumberOfPokemons`
  - ... but also for the entire class: `Pokemon.totalNumberOfPokemons`