



Programming in Java – Classes



Paolo Vercesi
Technical Program Manager

Agenda



Classes

instance variables, methods, ...

Constructors

default, multiple, the keyword “this”, ...

Initialization

Static

Equality and equivalence

A sample class

```
class Television {  
    String model;  
    boolean on;  
    int channel;  
    int volume;  
}
```

Definition of a class with four instance variables, *fields*

Creation of a new object of class Television

Declaration of a variable of type Television

```
new Television();  
Television tv;  
tv = new Television();
```

```
tv.model = "LG5464VX";
```

Instance variables are accessed using the *dot notation*

Creation and assignment of an object of class Television. The variable *tv* holds a *reference* to the new object.



Classes

Classes are used to define new types. Once a class is defined, we can then create new objects of that class. These objects are instances of that class, and that class is the type of these objects.

As *class* and *type* can be used interchangeably, so *object* and *instance* can be used interchangeably too.

A class is composed of

- *methods*, that define the operations that can be performed on its instances
- *instance variables*, that define the data that is associated to an object

Methods and instance variables are collectively known as *members*

Methods define how other entities can *interact* with the objects of that class.

Instance variables differentiate the behavior of different instances of the same class.



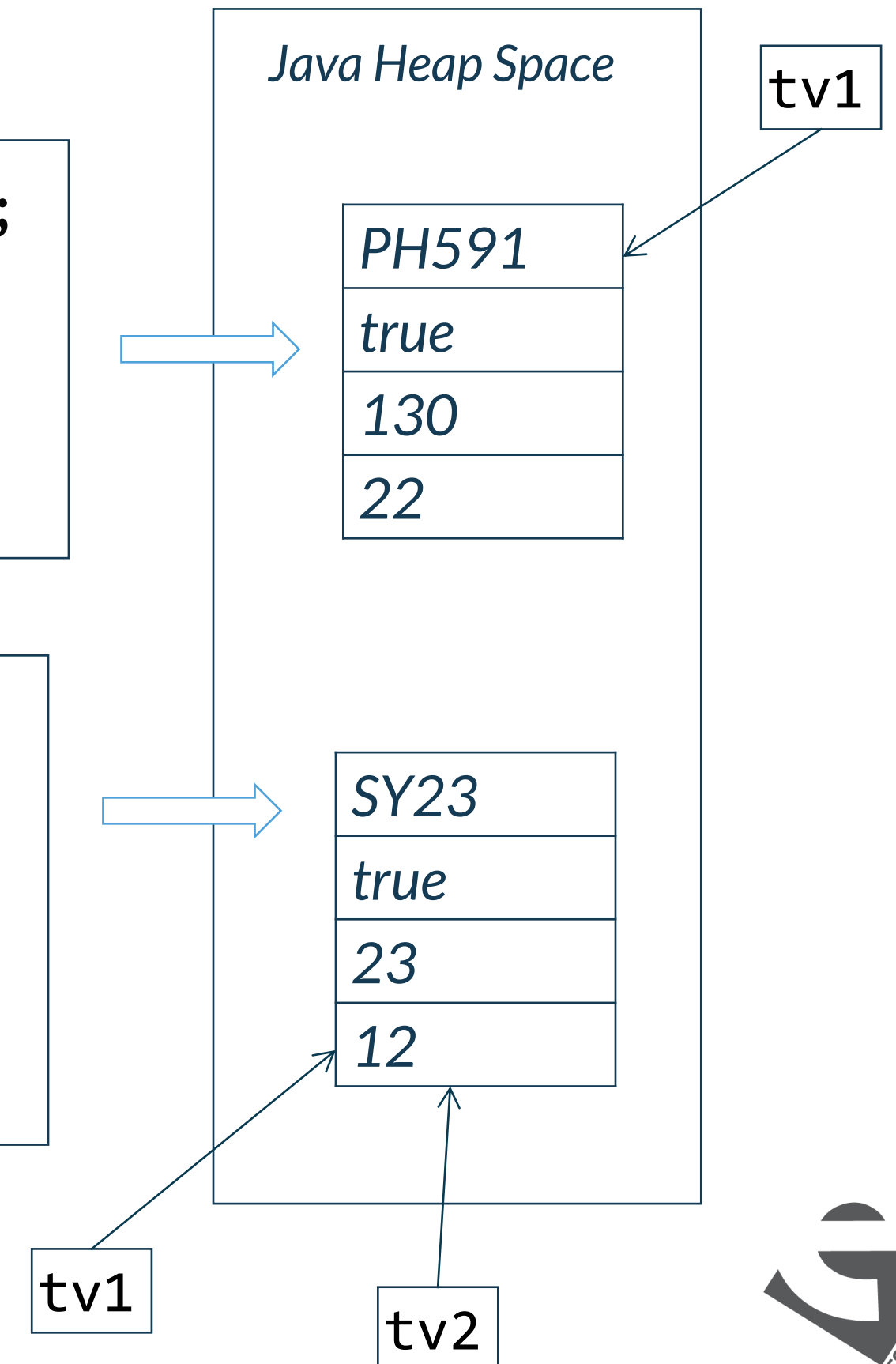
More on classes and references

When we create an object, using the *new* operator, we allocate the memory to hold the instance variables of this new object

```
var tv1 = new Television();  
tv1.model = "PH591"  
tv1.on = true;  
tv1.channel = 130;  
tv1.volume = 22;
```

Two, or more, *references* to the same object point to the same *memory location*.

```
Television tv1 = new Television();  
Television tv2 = tv1;  
tv1.model = "SY23"  
tv1.on = true;  
tv2.channel = 23;  
tv2.volume = 12;
```



Objects vs. primitive types or references vs. values

```
int i1 = 10;  
int i2 = i1; //we copy the value of i1 in i2  
i1 = i1 + 1; //now i1 is 11, i2 is still 10
```

Variables of primitive types hold the value represented by the primitive.

*Variables of **object references** hold a reference to the object*

No new object creation nor object copy is involved when we copy the reference from one variable to another

```
Television tv1 = new Television();  
Television tv2 = tv1;  
tv1.model = "SM2192"; //now both tv1 and tv2  
                      //model property is "SM2192"
```



Constructors

The creation of an object using the **new** operator is a two steps operation

1. Java allocates the memory for the object
2. the class constructor is invoked

If we don't define a constructor for a class, Java automatically defines a **default constructor**

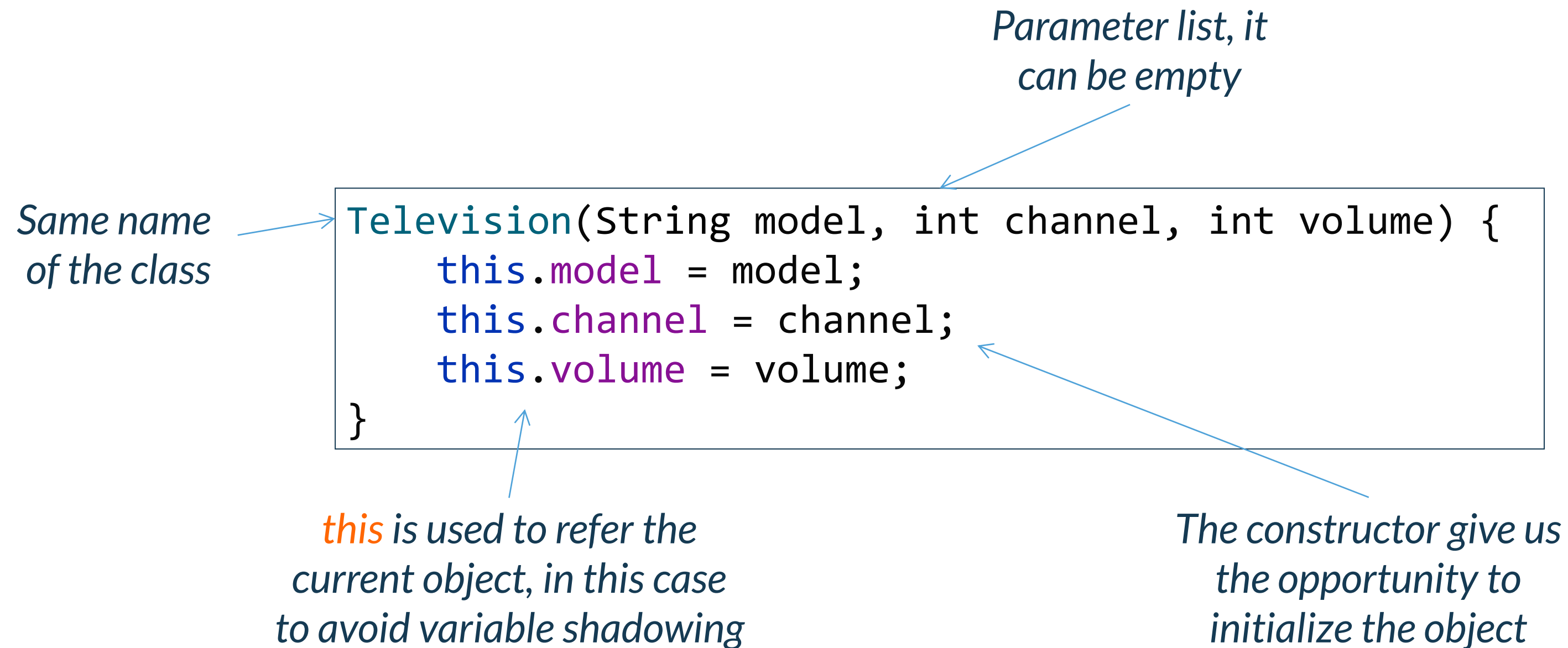
Parameter list, it can be empty

Same name of the class

```
Television(String model, int channel, int volume) {  
    this.model = model;  
    this.channel = channel;  
    this.volume = volume;  
}
```

this is used to refer the current object, in this case to avoid variable shadowing

The constructor give us the opportunity to initialize the object

The diagram illustrates a Java constructor for the Television class. The constructor signature is Television(String model, int channel, int volume). The body contains three assignment statements: this.model = model;, this.channel = channel;, and this.volume = volume;. Annotations with arrows point to specific parts: 'Parameter list, it can be empty' points to the parameter list in parentheses; 'Same name of the class' points to the class name Television; 'this is used to refer the current object, in this case to avoid variable shadowing' points to the 'this' keyword in the assignments; and 'The constructor give us the opportunity to initialize the object' points to the entire constructor block.

Constructor overloading

Java allows **constructor overloading**, a class can have multiple constructors, if their parameter lists are different.

The default constructor has an empty parameter list.

Once we define at least one constructor, the default one is no more available

```
class Television {
    String model;
    boolean on;
    int channel;
    int volume;

    Television(String model, int channel, int volume) {
        this.model = model;
        this.channel = channel;
        this.volume = volume;
    }

    Television(String model) {
        this.model = model;
    }
}
```



Constructor chaining

It is possible to invoke one constructor from another one, using `this()`, possibly with an argument list.

The call to `this()` must be the first statement within the constructor.

When `this()` is executed, the overloaded constructor that matches the parameter list is executed first. Then, if there are any statements inside the original constructor, they are executed.

```
class Television {
    String model;
    boolean on;
    int channel;
    int volume;

    Television(String model, int channel, int volume) {
        this(model);
        this.channel = channel;
        this.volume = volume;
    }

    Television(String model) {
        this.model = model;
    }
}
```



Construction of Strings

```
//Create a new String object
```

```
String a = "Some string";
```

```
//Create a new String object
```

```
String b = new String(new char[] {'S', 'o', 'm', 'e', ' ', 's', 't', 'r', 'i', 'n', 'g'});
```

```
//Create two String objects
```

```
String c = new String("Some string");
```

```
//Create a new String object
```

```
String d = a + b;
```

```
//We don't know, maybe the compiler is doing some optimization
```

```
String e = "Some" + " " + "string";
```



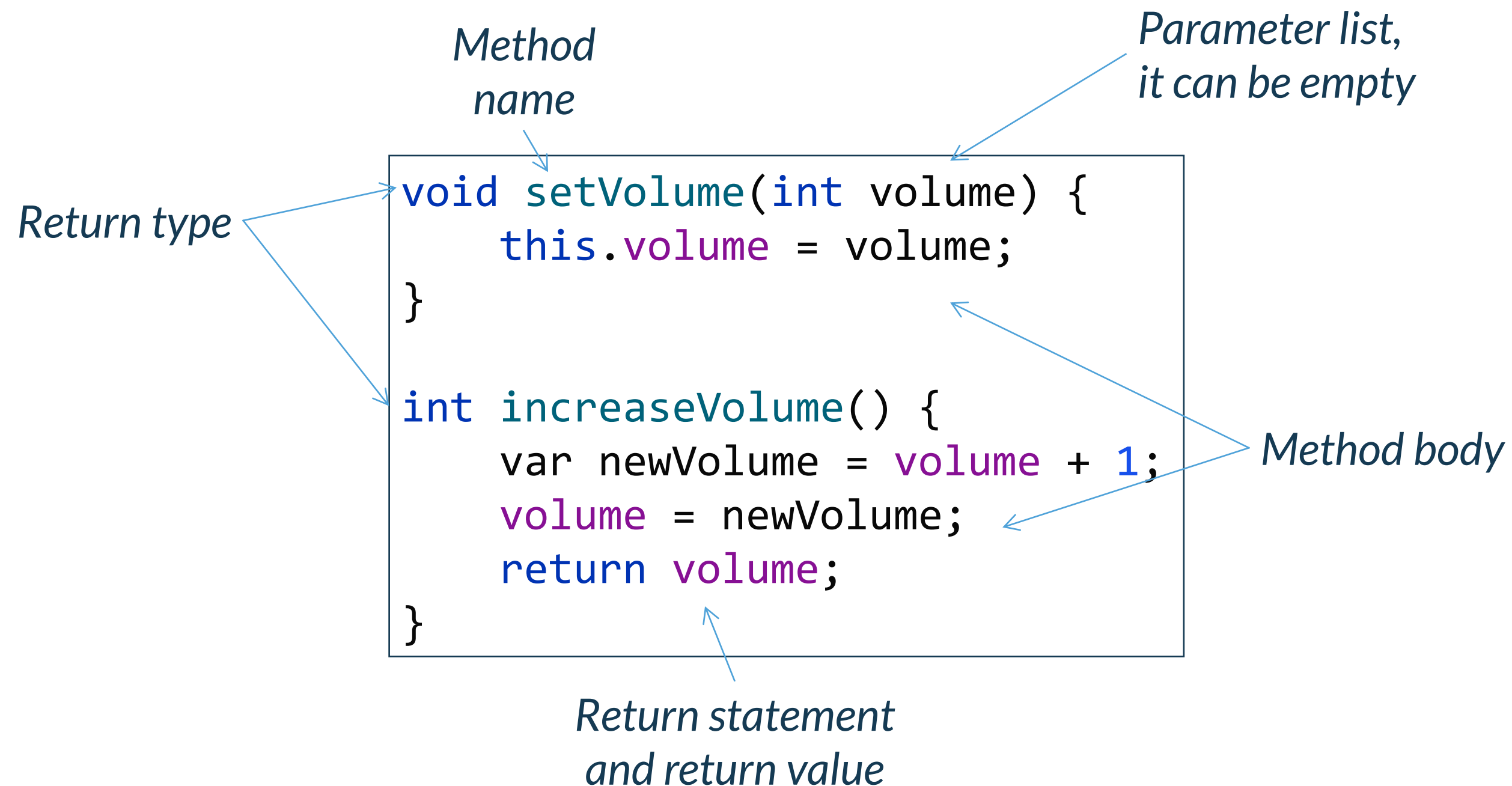
Destructors

*In Java there are **no** object destructors*



Methods

The Television class we have defined is almost useless. Why? It has no methods, it doesn't expose any behavior, it is just a data object. From an OOP perspective, data object are not good classes, if there are no methods operating on the data, it means that some other classes contain such methods to operate on the Television class data, this violates encapsulation.



Method invocation

Methods are invoked using the dot notation

```
Television tv = new Television("LG543");  
tv.turnOn();  
tv.setChannel(23);  
tv.increaseVolume(5);  
tv.setChannel(80);  
tv.turnOff();
```



Method overloading

Like the constructor case, Java allows **method overloading**, a class can have multiple methods with the same name, if their **parameter lists** are different.

```
int increaseVolume() {  
    return ++volume;  
}  
  
int increaseVolume(int delta) {  
    return volume += delta;  
}
```

Different
parameter list

Different
return type

```
int increaseVolume() {  
    return ++volume;  
}  
  
double increaseVolume() {  
    return ++volume;  
}
```

The **return type** cannot be used to differentiate overloaded methods

```
error: method increaseVolume()  
is already defined in class  
Television
```



Method chaining

```
class Television {
    String model;
    boolean on;
    int channel;
    int volume;

    Television(String model) {
        this.model = model;
    }

    Television setVolume(int volume) {
        this.volume = volume;
        return this;
    }

    Television turnOn() {
        on = true;
        return this;
    }

    Television turnOff() {
        on = false;
        return this;
    }

    Television setChannel(int channel) {
        this.channel = channel;
        return this;
    }
}
```

Using method chaining we can concatenate method invocations as in

```
new Television("ES213").turnOn()
    .setChannel(501).setVolume(16)
    .turnOff();
```



Methods with variable number of arguments

A variable length argument list is specified with three periods:

```
int add(int ... values) {  
    int summation = 0;  
    for (int i = 0; i < values.length; i++) {  
        summation += values[i];  
    }  
    return summation;  
}
```

```
int sum = add(1, 2, 3, 4, 5);
```

*The argument is implicitly declared as an **array**, however, the function can be called with a variable number of arguments*



The keyword “this”

The keyword *this* has two main uses

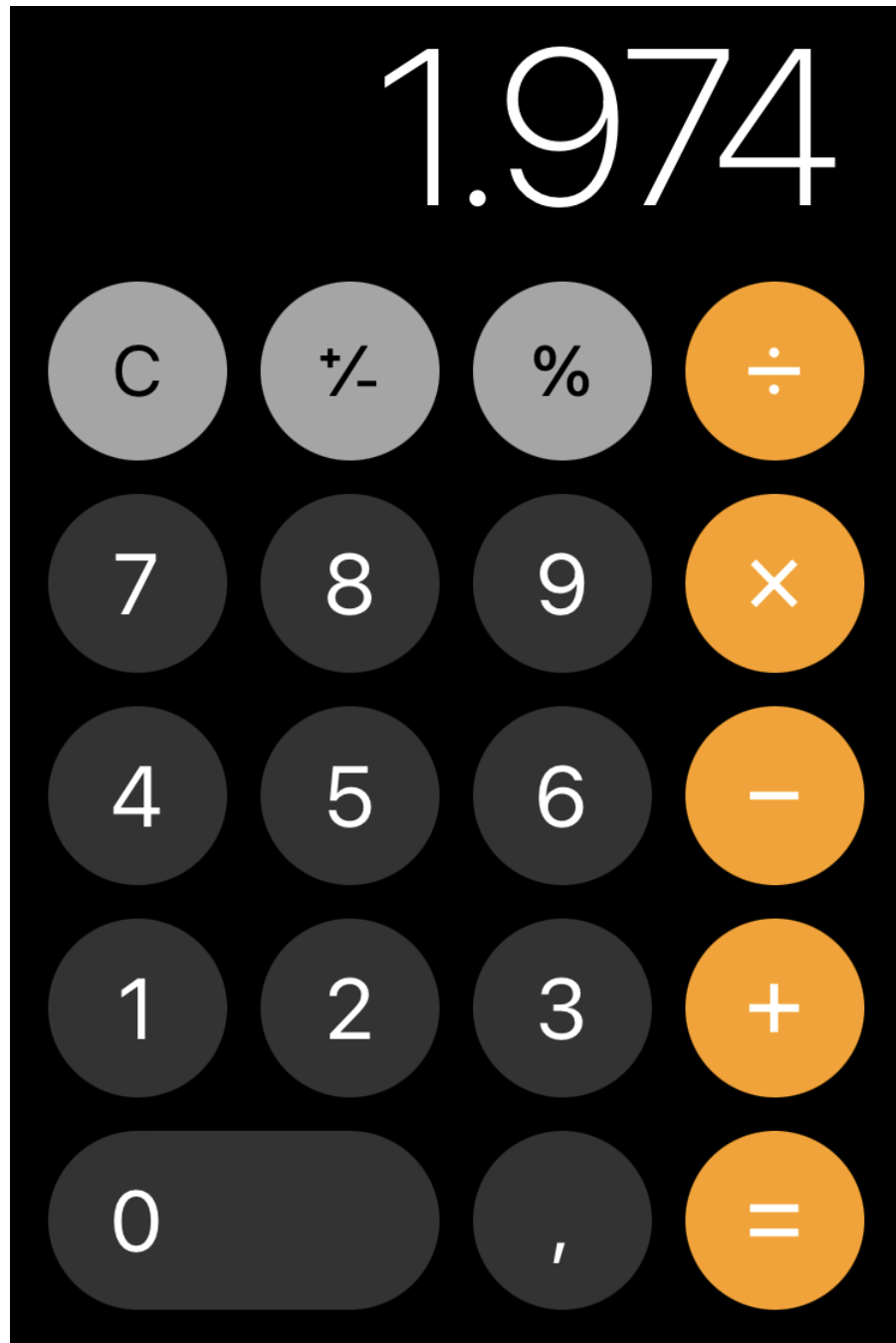
to *return a reference* to the current object

to *call constructors* from other constructors, constructor chaining

It has also other uses that we'll discover later



Assignment



Define a calculator class that

1. receives “events” from a calculator keyboard
2. sends the output to a *Display* object

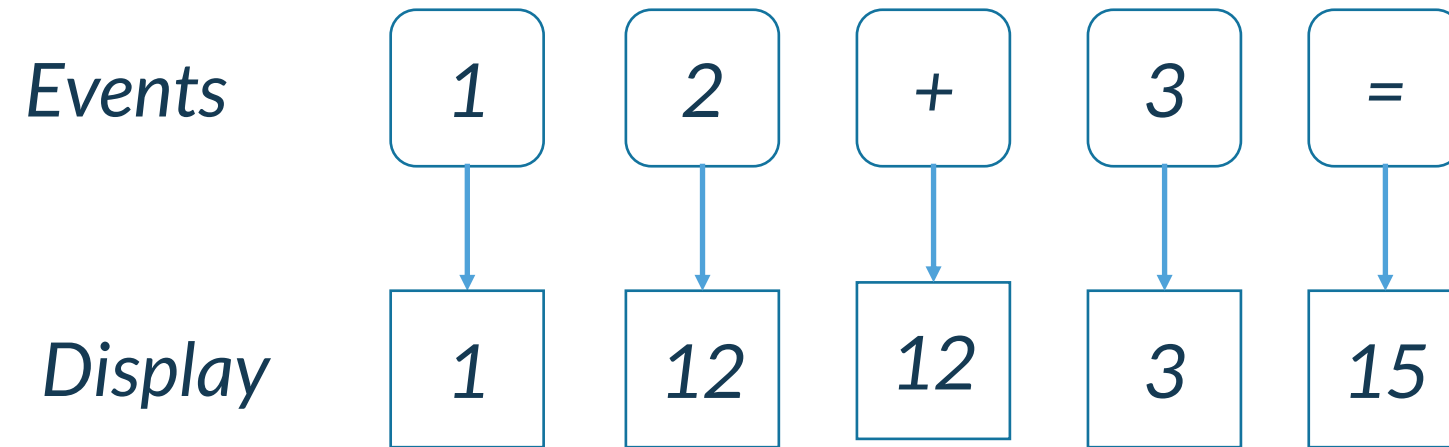
```
class Display {  
    void display(String text) {  
        System.out.println(text);  
    }  
}
```

```
class Calculator {  
    final Display display;  
    //...  
    Calculator(Display display) {  
        this.display = display;  
    }  
    void plusPressed() {  
        //...  
    }  
    void zeroPressed() {  
        //...  
    }  
    //...  
}
```

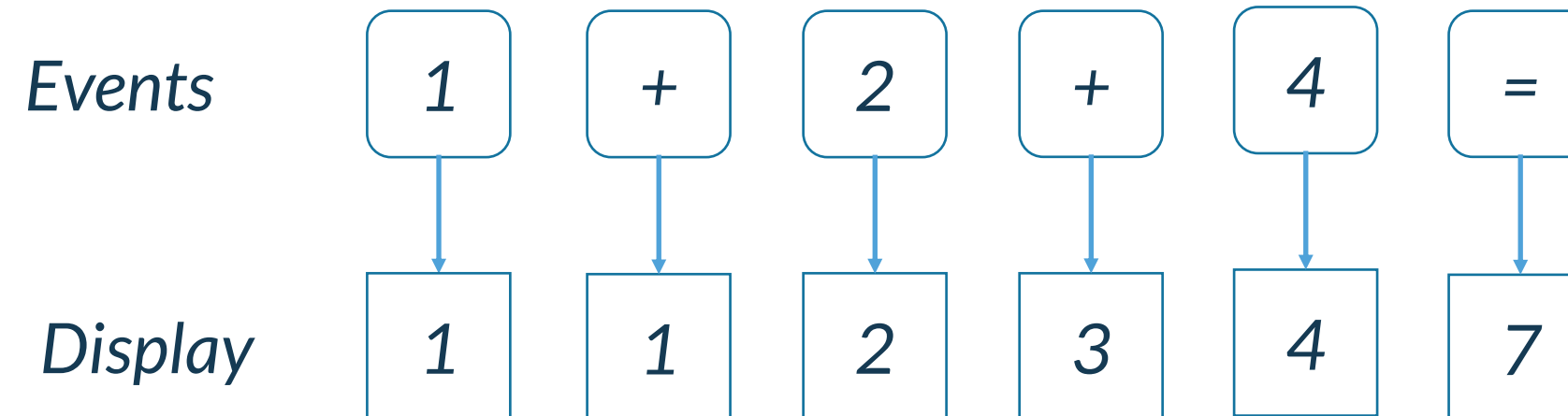


A couple of scenarios

Sample scenario

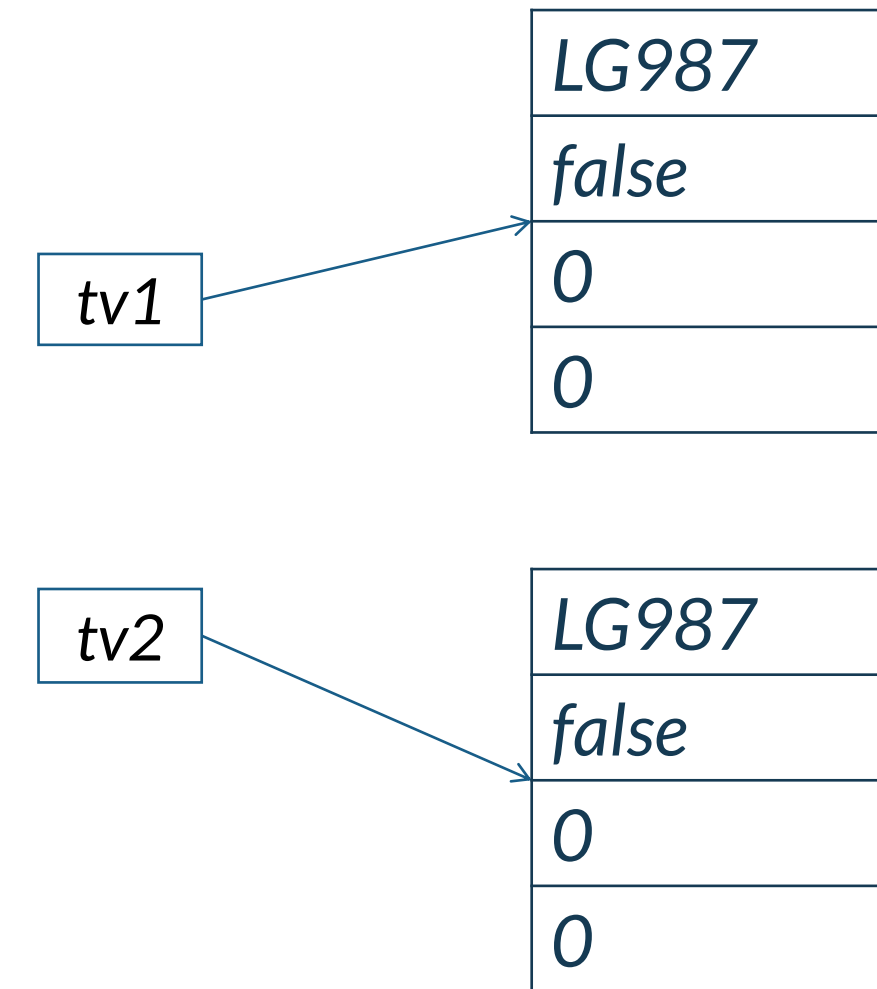


Operations chaining



Equality and equivalence 1/2

```
public static void main(String[] args) {  
    Television tv1 = new Television("LG987");  
    Television tv2 = new Television("LG987");  
  
    if (tv1 == tv2) {  
        System.out.println("Same");  
    } else {  
        System.out.println("Different");  
    }  
}
```



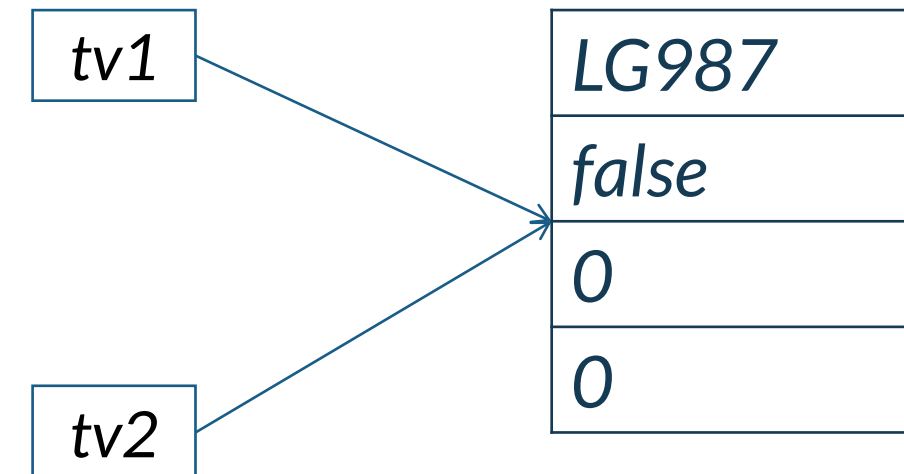
```
$ java Television  
Different
```

tv1 and tv2 are references to different objects



Equality and equivalence 2/2

```
public static void main(String[] args) {  
    Television tv1 = new Television("LG987");  
    Television tv2 = tv1;  
  
    if (tv1 == tv2) {  
        System.out.println("Same");  
    } else {  
        System.out.println("Different");  
    }  
}
```



```
$ java Television  
Same
```

tv1 and tv2 are references to the same object



Static members

HelloWorld.java

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

We have already met the *main* method that is declared *static* because it must be called before any object exists

Instance members (variables and methods) come to existence only when we create objects of their classes.

On the contrary, when a member is declared *static*, it is a *class member* and it can be accessed without any reference to objects of its class.

The keyword *static* must precede the member declaration.



Static variables

Variables declared as **static** are a sort of global variables.

When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Fields that are both **static** and **final** can be used as global constants.

```
class Television {  
    static int numberOfTelevisionsTurnedOn;  
  
    String model;  
    boolean on;  
    int channel, volume;  
  
    Television(String model) {  
        this.model = model;  
    }  
  
    void turnOn() {  
        on = true;  
        numberOfTelevisionsTurnedOn++;  
    }  
  
    void turnOff() {  
        on = false;  
        numberOfTelevisionsTurnedOn--;  
    }  
}
```



Static method restrictions 1/3

Static methods can only directly invoke other *static methods*, they cannot invoke *instance methods* if they haven't a reference to an object

```
class Television {  
    int volume;  
  
    static Television increaseVolumeBy5() {  
        increaseVolume(5);  
    }  
  
    Television increaseVolume(int delta) {  
        volume += delta;  
    }  
}
```

```
class Television {  
    int volume;  
  
    static void increaseVolumeBy5(Television tv) {  
        tv.increaseVolume(5);  
    }  
  
    int increaseVolume(int delta) {  
        return volume += delta;  
    }  
}
```

error: Non-static method
'increaseVolume(int)' cannot be referenced
from a static context



Static method restrictions 2/3

Static methods can only directly access *static variable*, they cannot access *instance variables* if they haven't a reference to an object

```
class Television {  
    int volume;  
  
    static Television increaseVolumeBy5() {  
        volume += 5;  
    }  
  
    Television increaseVolume(int delta) {  
        volume += delta;  
    }  
}
```

```
class Television {  
    int volume;  
  
    static void increaseVolumeBy5(Television tv) {  
        tv.volume += 5;  
    }  
  
    int increaseVolume(int delta) {  
        return volume += delta;  
    }  
}
```

error: Non-static field 'volume' cannot be referenced from a static context



Static method restrictions 3/3

Static methods cannot refer to *this* in any way

```
class Television {  
    int volume;  
  
    static Television increaseVolumeBy5() {  
        this.volume += 5;  
    }  
  
    Television increaseVolume(int delta) {  
        volume += delta;  
    }  
}
```

```
class Television {  
    int volume;  
  
    static void increaseVolumeBy5() {  
        this.increaseVolume(5);  
    }  
  
    int increaseVolume(int delta) {  
        return volume += delta;  
    }  
}
```

error: Television.this cannot be referenced from a static context



Accessing static members

```
class TelevisionMain {  
    public static void main(String[] args) {  
        Television.setInitialVolume(5);  
        System.out.println("Number of televisions turned on: " +  
            Television.numberOfTelevisionsTurnedOn);  
    }  
}
```

Outside of the class in which they are defined, *static methods* and *static variables* can be used independently of any object. To do so, you need only to specify the *name of their class followed by the dot operator*.



Resolving shadowing of static variables

```
class Television {  
  
    static int initialVol;  
  
    int volume;  
  
    static void setInitialVolume(int initialVol) {  
        initialVol = initialVol;  
    }  
  
    Television() {  
        volume = initialVolume;  
    }  
}
```

```
class Television {  
  
    static int initialVol;  
  
    int volume;  
  
    static void setInitialVolume(int initialVol) {  
        Television.initialVol = initialVol;  
    }  
  
    Television() {  
        volume = initialVolume;  
    }  
}
```



Default values

TelevisionMain.java

```
class Television {
    static int initialVol;
    String model;
    boolean on;
    int channel;
    int volume;
}

class TelevisionMain {
    public static void main(String[] args) {
        Television tv = new Television();
        System.out.println(tv.model);
        System.out.println(tv.on);
        System.out.println(tv.channel);
        System.out.println(tv.volume);
        System.out.println(Television.initialVol);
    }
}
```

All *fields*, both instance variables and class variables, are *guaranteed* to have an initial value when we create an object

Object references are initialized with the *null reference*, numeric variables with *0*, boolean variables with *false*

So, in principle we don't need to initialize fields, we only need to initialize *local variables* before their use

```
$ java TelevisionMain.java
null
false
0
0
0
```



Revisiting System.out.println

HelloWorld.java

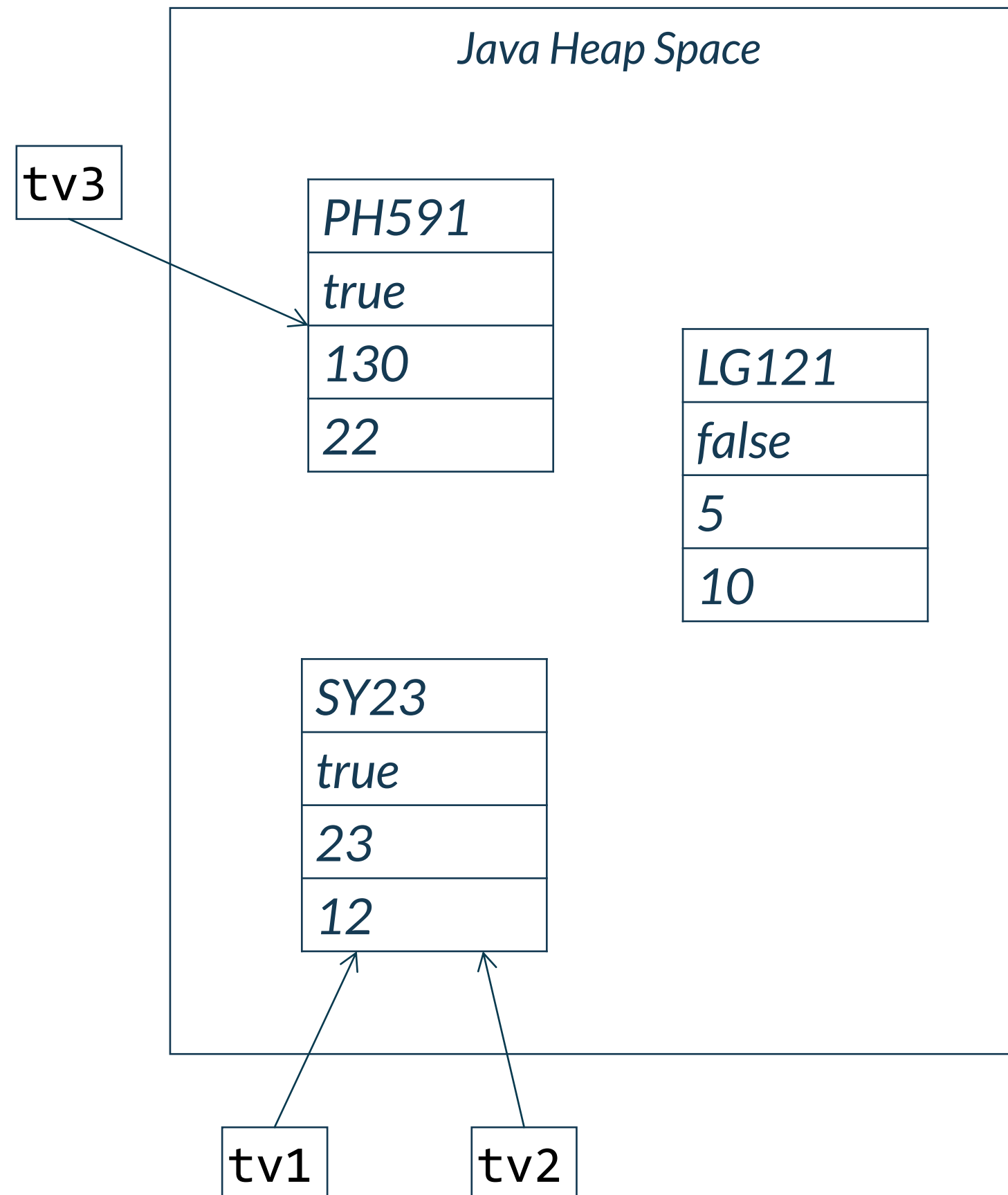
```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/System.html#out>



A word about garbage collection

```
var tv1 = new Television();  
Television tv2 = tv1;  
tv1.model = "SY23"  
tv1.on = true;  
tv2.channel = 23;  
tv2.volume = 12;  
  
var tv3 = new Television();  
tv3.model = "PH591"  
tv3.on = true;  
tv3.channel = 130;  
tv3.volume = 22;  
  
var tv4 = new Television();  
tv4.model = "LG121"  
tv4.on = false;  
tv4.channel = 5;  
tv4.volume = 10;  
  
tv4 = null;
```



Memory locations that are not referenced in any way are eligible to be **garbage collected**

The memory of garbage collected objects is freed and made again available in the Heap Space





Thank you!

esteco.com

