

# Introduction to Programming

## Part I

### Lecture 7 Introduction to Java

Eugene Zouev  
Fall Semester 2021  
Innopolis University

# The First Look at Java: What We Have Learnt

- **Object-oriented approach** to programming: basic idea (to be discussed in details later)
- **Classes**: what's this and how to declare them
- **Class instances** (objects): how to create them
- **Value types** and **reference types**
- Class instances as pairs of the instance itself and the reference to it
- Access to instances: dot notation
- **Access control**: public and private members
- Destroying instances: automatic **garbage collection**
- **Constructors**

# The Structure of Java Programs

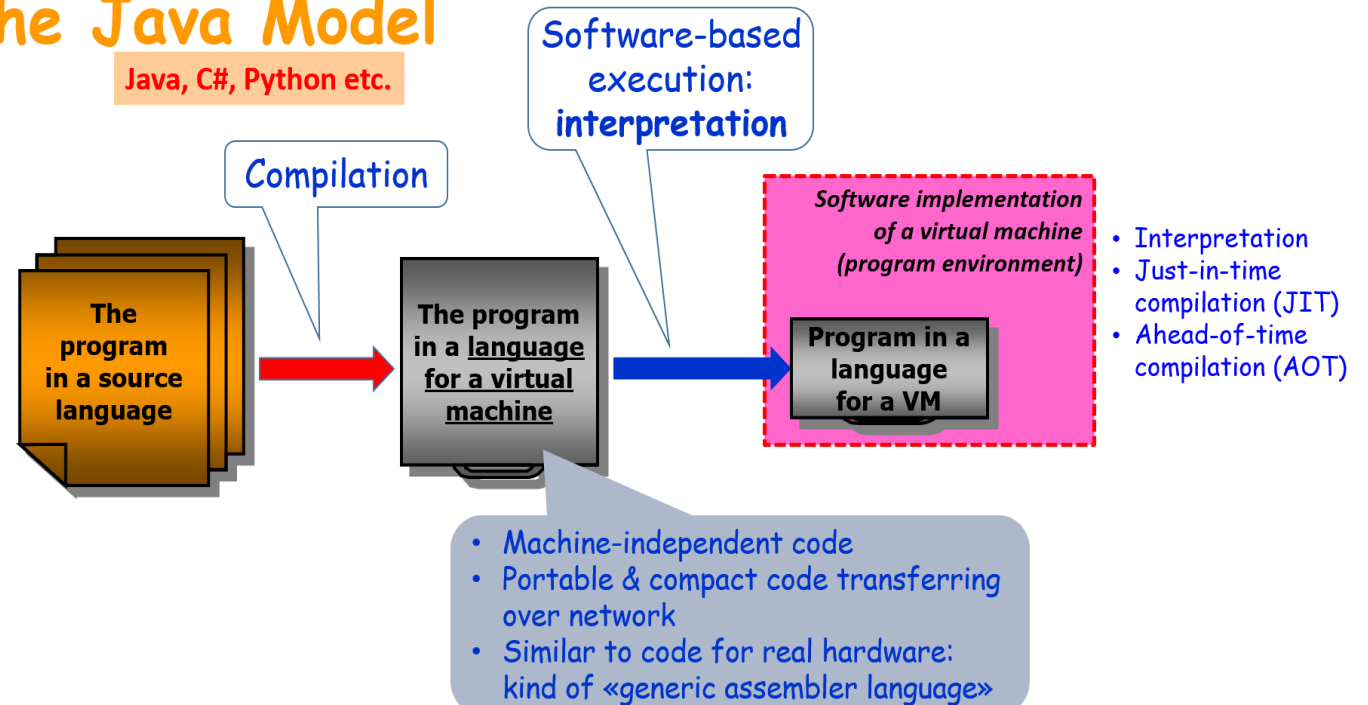
- Java program is a collection of **classes**
- **Class** is the main program building block, and the key notion of **object-oriented programming**
- In general, class has many important features (*later we will consider them all carefully*), but all you have to know for today is:

**Class** is a language construct comprising **algorithms** (in form of **functions**) and **data** the algorithms work on

*Simplified*

## Compilation & Execution: the Java Model

Java, C#, Python etc.



# Class Example

**Class is a (user-defined) type**

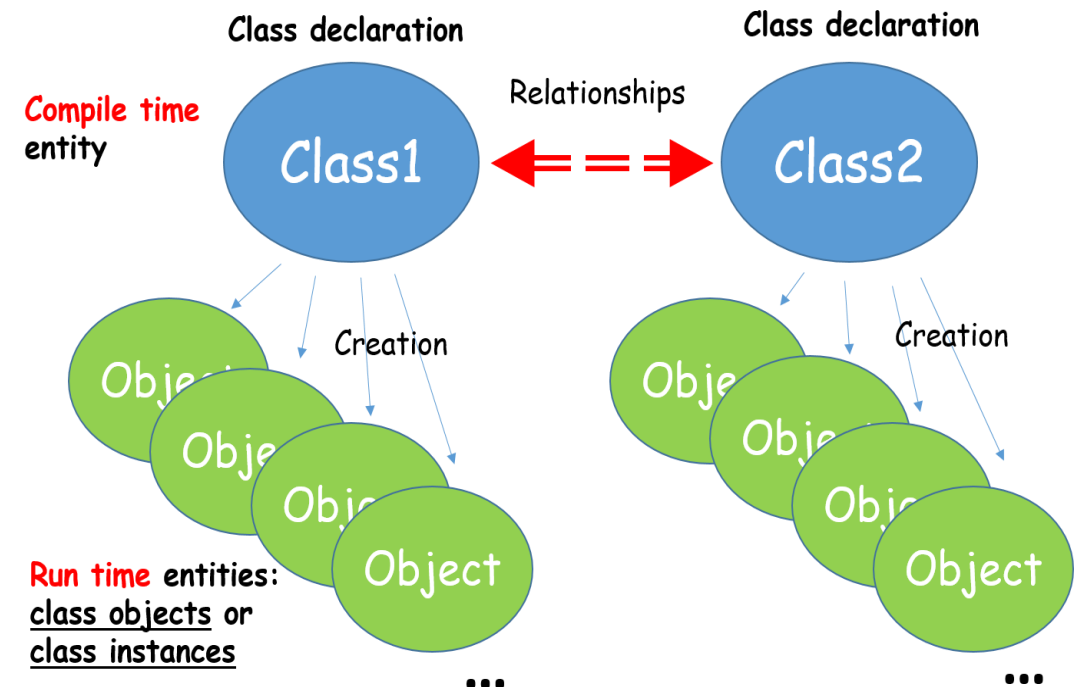
In general, class should **completely specify all aspects** of objects that are created by this class:

- The **state** of class objects
- The **behavior** of class objects
- The way of **creating** objects
- The way of **destroying** objects (when/if they are not needed anymore)
- **Relationships** between this object and other objects of the same class or of some other class(es)

Class declaration specifies *pattern*. Objects will be created using this pattern.

```
class Point
{
    int x;
    int y;
    void move(int dx, int dy)
    {
        x += dx;
        y += dy;
    }
}
```

## Classes & Objects



Class specifies a pattern (a template, an example) for creating real entities of the class: they are called **instances**, or **objects** of the class.

# Constructors

OR: How to initialize class instances

## Constructor:

The special method whose name is the same as the class name. It's automatically called by the **new** operator.

There can be several constructors defined for a class. The idea is that a class developer can provide several ways for creating instances.

```
class SomeOtherClass
{
    Point p1 = new Point();
    Point p2 = new Point(3,4);
}
```

```
class Point
{
    int x, y;

    public Point()
    {
        x = 0; y = 0;
    }
    public Point(int a1,
    {
        x = a1; y = a2;
    }
    public void move(int
    {
        x += dx;
        y += dy;
    }
}
```

Constructors here are made **public**: they are treated as a part of class interface

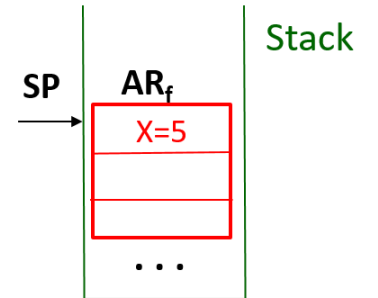
## Value and Reference Types

- There are two categories of types in Java: **value types** and **reference types**.

Examples of value types: integers, floating, doubles. Values of these types are represented directly:

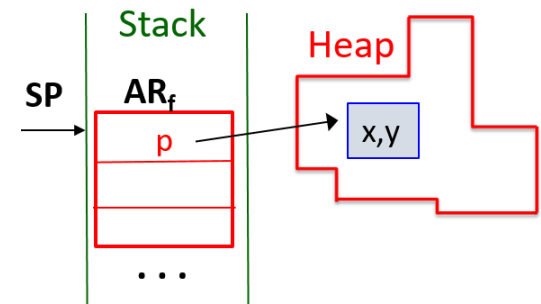
- Classes are reference types**. This means instances of classes always exist as **pairs**: the instance itself and the representative of the instance - the **reference**:

```
int x = 5;
```



```
Point p = new Point();
```

Internally, **p** is just an **address (pointer)** of the instance in the heap...



# What's For Today

- More on interface & implementation
- More on constructors
- How to pass parameters to methods
- Class attributes & class methods
- The method `main`
- Java packages

# Interface & Implementation

## B.Stroustrup about class interfaces:

- Interface should be complete
- Interface should be minimal
- Class should have constructors
- Class should support copying - or should explicitly prohibit it
- Careful argument checks should be provided
- Destructor should make all resources free

# The Information Hiding Principle

- The designer of a class must specify which properties are accessible to clients (i.e. public) and which are internal (hidden).
- The programming language must ensure that clients can only use public properties.

## Encapsulation:

the first cornerstone of the object-oriented approach.



# Multiple Constructors

```
class Point
{
    int x, y;

    public Point() {
        x = 0; y = 0;
    }
    public Point(int x0, int y0) {
        x = x0; y = y0;
    }
}

class OtherClass
{
    void f() {
        Point p1 = new Point();
        Point p2 = new Point(1,2);
    }
}
```

Several constructors

- The idea is to provide users **several ways for creating objects**.
- Constructor without parameters is called **default constructor**.

# Refactoring Constructors

```
class Point
{
    int x, y;

    public Point() {
        x = 0; y = 0;
    }
    public Point(int x0, int y0) {
        x = x0; y = y0;
    }
    public Point(int d) {
        this(d,d);
    }
}

class OtherClass
{
    void f() {
        Point p1 = new Point();
        Point p2 = new Point(1,2);
        Point p3 = new Point(5);
    }
}
```

- Constructors in Java can **call other constructors** of the same class
- This can be done by using the keyword **this**
- It allows to factor out common behaviors

# Constructors and "Inline initialization"

- Java allows the specification of default values of the attributes on the line of their declaration.
- It's called "inline initialization" of the attributes

```
class Point
{
    int x = 0;
    int y = 0;

    public Point(int x0, int y0) {
        x = x0; y = y0;
    }
}
```

When an object is created:

- The inline initialization of the attributes is performed (if any) **in the order of the appearance** of the attributes in the code.
- The constructor is called and its actions are executed.

# Function Parameters

Why parameters?

✓ To make functions/methods more useful for more than one use case.

```
int key()
{
    return 77;
}
```

BTW: are these  
functions  
useless?

```
void printKey()
{
    System.out.println(key());
}
```

```
class C
{
    int x;
    public int getX() { return x; }
}
```


```
int inc(int v)
{
    return v+1;
}
```

```
void printValue(int v)
{
    System.out.println(v);
}
```

```
int sqr(int v)
{
    return v*v;
}
```

# Function Parameters

- Formal parameters (or just **parameters**)
  - Used to define functions



```
int sqr(int v)
{
    return v*v;
}
```

Usually (in most programming languages) parameter looks like a **variable declaration**

- Actual parameters (or **arguments**)
  - Used when the function is called



```
int q5 = sqr(5);
```

Usually (in most programming languages) argument is an **expression**

# Function Parameters

## Common mechanism:

- When a function is called formal parameters in the function declaration **are replaced** for arguments taken from the call.
  - That is, formal parameters get values from corresponding arguments.
- The function body is executed
  - That is, function's statements are executed using actual values.

What does it mean exactly?

There are two main ways of passing parameters:

- **By value**
- **By reference**

# Parameter Passing

- **BY VALUE**

The value of the actual parameter is copied to the corresponding formal parameter

- Concretely, into the stackframe of the called function to the slot reserved for the formal parameter

😊 Safe: a formal parameter is an **independent copy** of an actual parameter

😞 Cumbersome: for complex data structure

# Parameter Passing

- **BY REFERENCE**

A **reference** to the actual parameter is copied to the corresponding formal parameter.

- Concretely into the activation record of the called function to the slot reserved for the formal parameter

☺ Improves efficiency: when a formal parameter is changed in the procedure an actual parameter changes too → they both refer to the same entity

☹ Not safe: functions may have annoying and dangerous side effects on their parameters



# Parameter Passing in Java

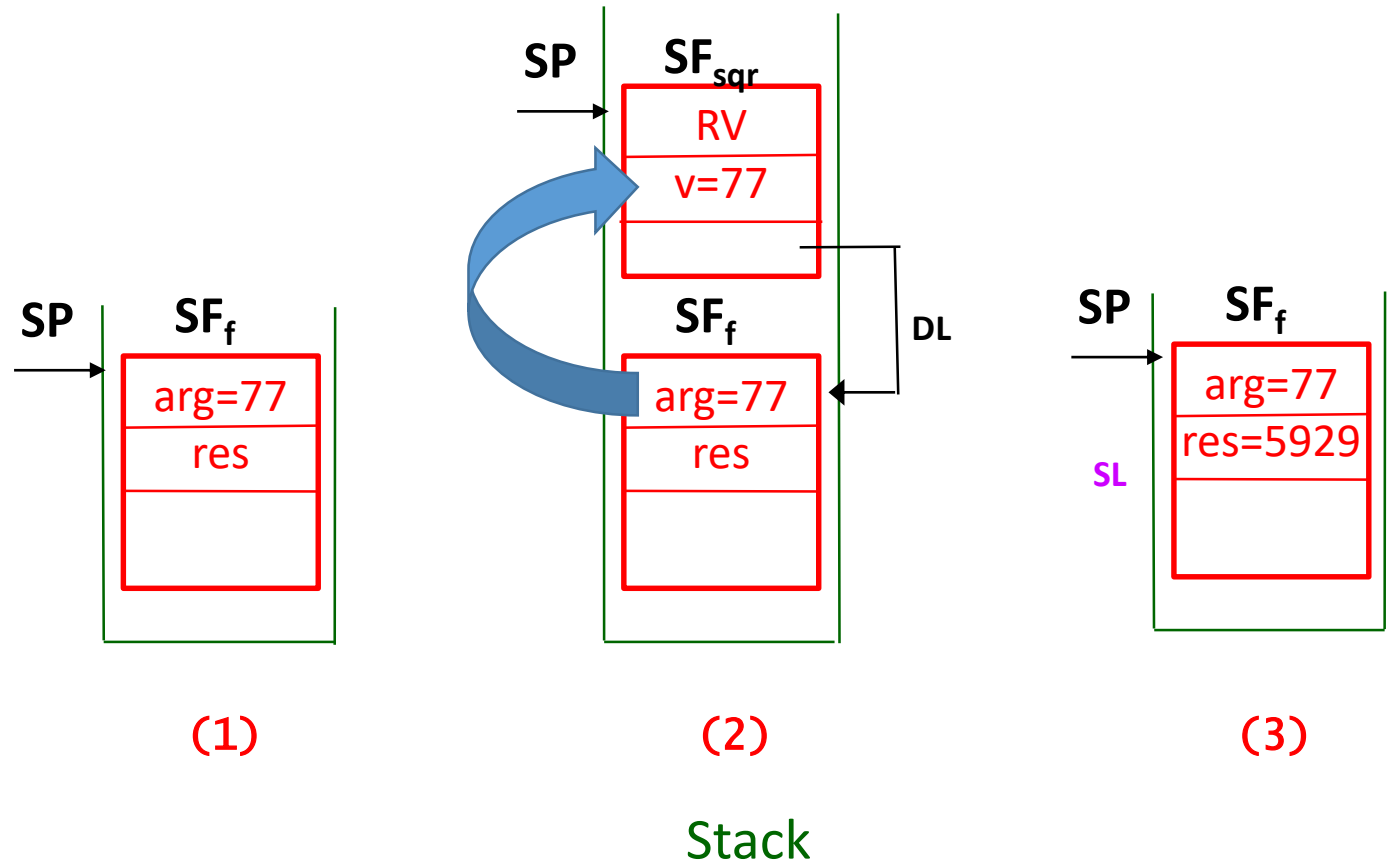
- Parameters of **value types** are always passed **by values**
  - That is, integers, doubles etc are just copied to formal parameters.
- Parameters of **reference types** are passed **by reference**
  - That is, class instances are **not passed**, but their references are.

Passing a reference to an object has the side effect that the referenced object can be modified.

# Parameter Passing in Java: Example

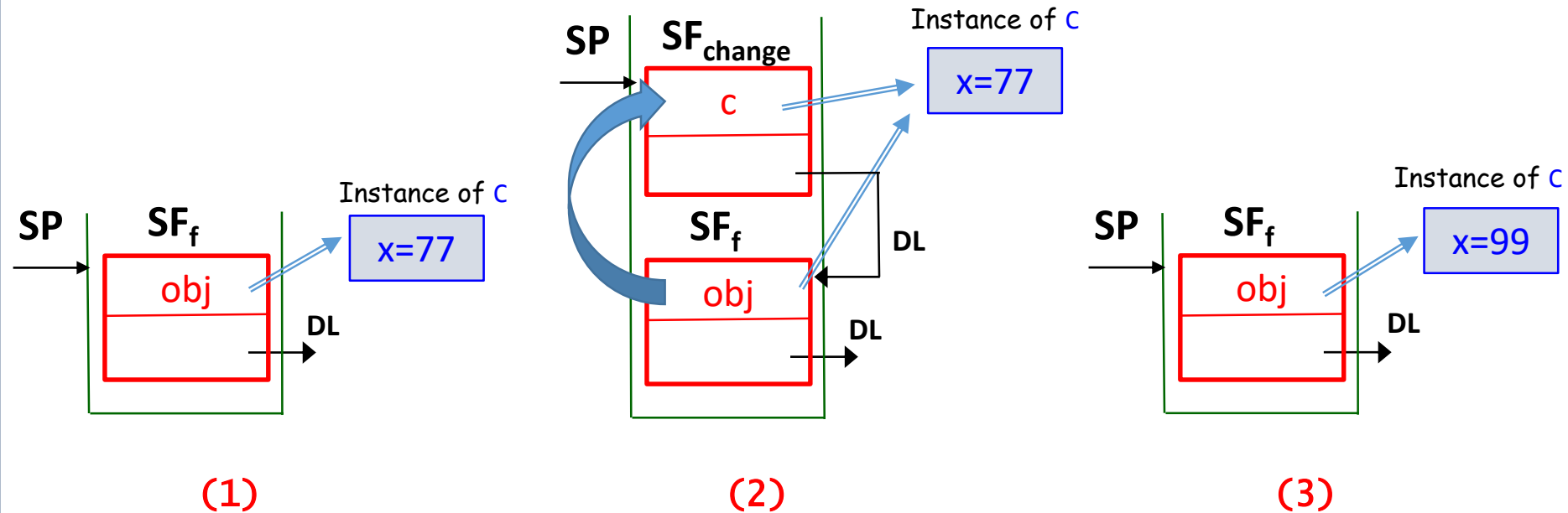
```
class Example
{
    int sqr(int v)
    {
        return v*v;
    }

    void f()
    {
        (1) int arg = 77;
        (2) int res = sqr(arg);
        (3) ...
    }
}
```



# Parameter Passing in Java: Example

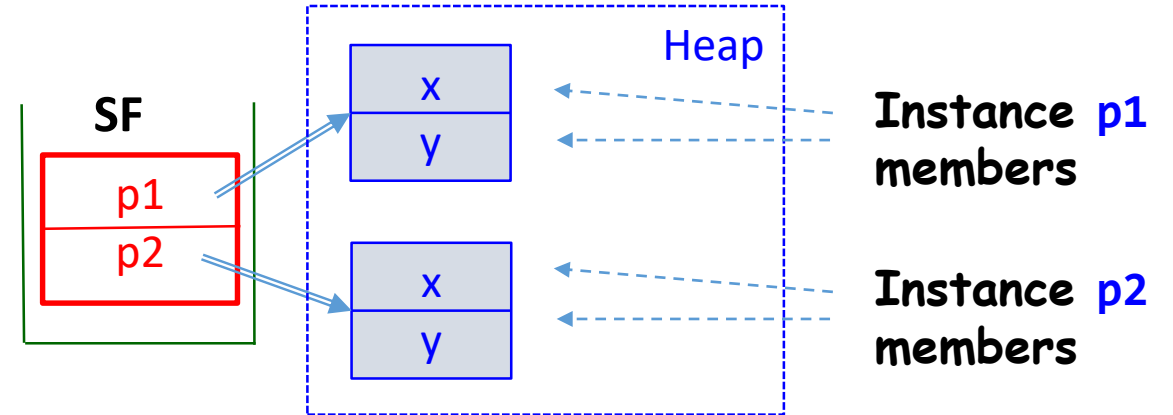
```
class C {  
    public int x;  
}  
class Example {  
    void change(C c)  
    {  
        c.x = 99;  
    }  
    void f()  
    {  
        C obj = new C();  
(1) obj.x = 77;  
(2) change(obj);  
(3) ...  
    }  
}
```



# Class Attributes

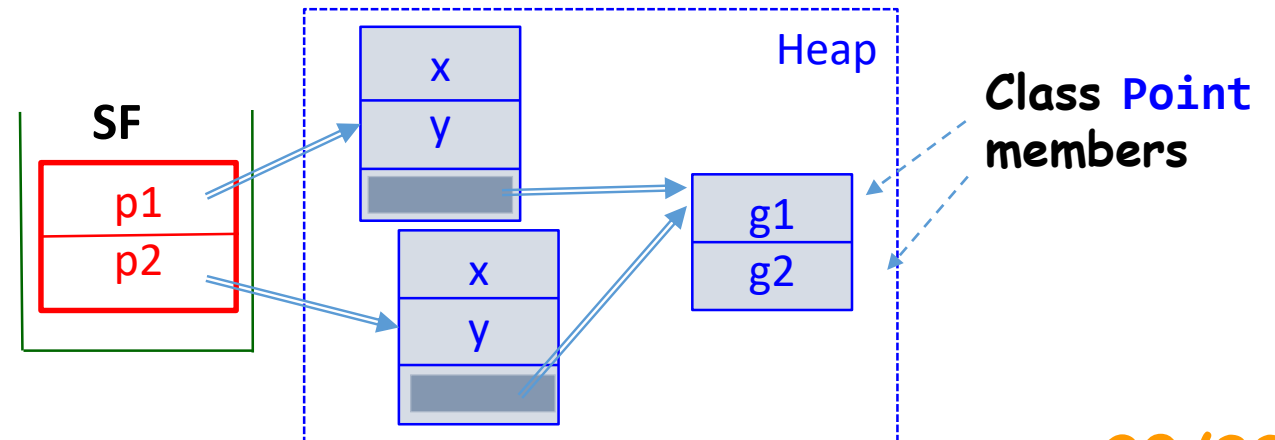
- Usually, each instance has **its own set of attributes**

```
class Point
{
    int x, y;
}
...
Point p1 = new Point();
Point p2 = new Point();
```



- Class attributes:** **belong to the class as a whole** and are **shared among all the objects** of the class

```
class Point
{
    int x, y;
    static int g1, g2;
}
...
Point p1 = new Point();
Point p2 = new Point();
```



# Class Attributes

- Class attributes do not belong to a particular instance but belong to all instances of the class - or, they "belong to a class as a whole".
- The Java jargon refers to them with the term "**static attributes**".
- Class attributes cannot be stored in an object, as they are shared among all objects that are instances of the same class.
  - Java class attributes are stored in an area of the heap specifically devoted to them → "**area for statics**"
- Scope: the same as the scope of the class they belong to.

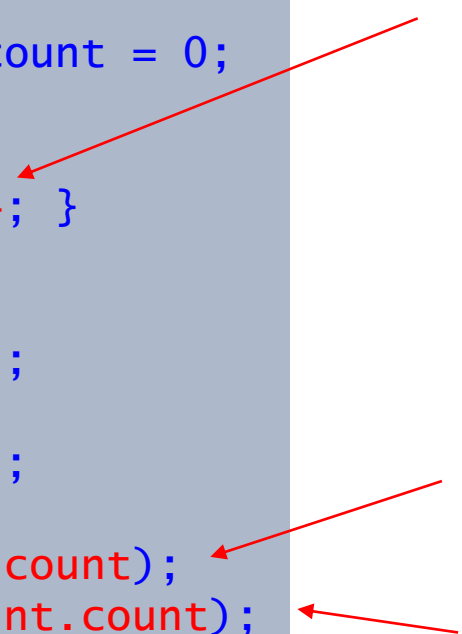
# Class Attributes

## Typical use case

```
class Point
{
    int x, y;
    public static int count = 0;

    public Point()
    { x=0; y=0; count++; }
}

...
Point p1 = new Point();
...
Point p2 = new Point();
...
System.out.println(p2.count);
System.out.println(Point.count);
```



When each instance of **Point** is created, **count** gets increased by one. Therefore, at any time while program runs, **count** contains the overall number of instances created.

Access to static attributes can be performed by usual **dot notation**:

- Either using the name of (any) instance of the class, OR
- Using the name of the class itself (recommended).

# Class Methods

- **Class (“static”) methods** are methods that can be invoked by a class, not by an instance.

OR:

these methods are associated **directly with the class.**

- Static method is used in most cases when there is a standalone utility method that should be made available without requiring the overhead of instantiation

# Class Methods

- Class methods do not need objects to be invoked.
- A class methods cannot access object attributes, that is, attributes that refer to a specific object instance of the class; they can access only to **class attributes**.
  - Hence, a class method doesn't have hidden **this** parameter
- Class methods can be invoked - by the usual dot notation - either via the class name or via (any) instance name of that class. Its behavior is the same in both cases.
- Class method can be invoked even when **no one instance** has been created for the class.



# Example of the Class Method

```
class Point
{
    int x, y;

    static int max_x = 100;
    static int max_y = 100;

    static void check(int x, int y)
    {
        if (x>max_x || y>max_y)
            throw maxError;
    }

    public Point(int x0, int y0)
    { check(x0,y0); x=x0; y=y0; }
}

...
Point p1 = new Point(2,3);    // OK
Point p2 = new Point(20,300); // exception
```

The idea is to define **the limits** of our two-dimensional plane where we allocate points. These limits are common to all points therefore we made them **static**.

The class method **check** checks whether coordinates of a point are within limits. If any of coordinate exceeds the limit we "throw an exception" (will discuss exception mechanism later).  
Again, this method is common to all points created while the program runs.

We apply **check** for each new point.

Both class attributes **max\_x**, **max\_y** and class method **check** are private by default because they are to be used only within the class.

# main: The Special Class Method

The question: how the JVM executes a Java program?

- If the Java Virtual Machine detects a class in a program that contains **public static method** called **main** - then the JVM treats this method as the **starting point** ("entry point") of the program.
  - The class containing the **main** method can have any name.
- To **main** method accepts parameter which is of type **String[]** - that is, the array of strings.
  - Simply speaking, the **main** method can be invoked with any number of arguments that are treated as strings.

# main: An Example

Program.java

```
class Point
{
    int x, y;
    ...
}
```

Any class can  
contain `main`

→ `class Program`

```
{
    public static void main(String[] pars)
    {
        int x0 = pars[0].toInt();
        int y0 = pars[1].toInt();
        Point p = new Point(x0,y0);
    }
}
```

← Program entry point

Console

```
>javac Program.java
>java Program 7 8
```

Ask your TA about how to compile and run your programs, about the meaning of the `String` type and `toInt()` method, and how to get the number of arguments passed to `main`.

# null: The Empty Reference

```
Point p = new Point(1,2);
```

1. This is declaration; the variable `p` is declared.
2. The type of `p` is the class type `Point`.
3. The declaration contains initialization; the value of the expression from the initialization becomes the initial value of `p`.
4. The value of the `new` expression is the reference to the newly created instance of class `Point`.

```
Point p;
```

***What's this?***

1. This is declaration; the variable `p` is declared.
2. The type of `p` is the class type `Point`.
3. The declaration doesn't contain initialization; this means that the initial value of `p` is the empty reference.
4. The empty reference is denoted as `null`.

```
Point p = null;
```

# this : The Special Reference

- **this** is the reference used for getting access to instance members from within method bodies
- By definition, **this** is the reference to the instance for which the function is called.
- **this** can be used only within function member bodies.

```
class C {  
    int member;  
    public void f(int i)  
        { member = i; }  
};
```

By definition, these two constructs are semantically equivalent:

```
public void f(int i) { member = i; }
```

```
public void f(int i) { this.member = i; }
```

Syntactically, **this** is the keyword

# this : The Special Reference

How a method “knows” about the instance for which it is called?

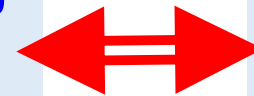
- The pointer to instance gets passed to the function as very first hidden parameter!!
- Just *this* hidden parameter is known within the function as **this**.

```
class C {  
    public void f(int i)  
    { ... }  
};
```

(This is semantic equivalence; it's not correct syntactically)

```
C c = new C();  
c.f(1);
```

The call to a method  
is treated by the  
compiler as...



```
C c = new C();  
f(c, 1);
```

...passing the reference  
to the instance as the  
first parameter

# The First “Real” 😊 Java Function

```
int gcd(int x, int y)
{
    int a = x, b = y;
    while ( a != 0 )
    {
        int temp = a;
        a = b % a
        b = temp
    }
    return b;
}
```

**Euclid algorithm:**  
Finds the greatest common  
denominator for two numbers  
Наибольший общий делитель

Imperative paradigm

Some important points:

- The algorithm is organized as a series of **steps**.
- The variables **change their values** on each step.
- There are **three local variables** used in the algorithm.
- This is the **iterative** algorithm (with loop).

**Is it the best implementation of the Euclid algorithm?**

# The First "Real" Java ☺ Function

```
int gcd(int x, int y)
{
    int a = x, b = y;
    while ( a != 0 )
    {
        int temp = a;
        a = b % a;
        b = temp;
    }
    return b;
}
```



```
int gcd(int x, int y)
{
    if (y == 0)
        return x;
    else
        return gcd(y, x%y);
}
```

Can we make the function even more compact? ☺

Functional paradigm

Important points:

- **No** local variables.
- Variables (parameters) do not **change** their values.
- This is the **recursive** algorithm: recursion is used instead of iteration
- The code is much more concise and readable.



# The First “Real” Java Program

Methods for the standard class `InputStream` will be used in the program (`read`)

*Not mandatory in this case!*

The main method: Java programs always start execution from it. It should present in any class of the program.

`read` reads a short value from the console; `println` outputs its argument to the console.

```
import java.io.InputStream

public class Program {

    int gcd(int x, int y)
    {
        return (y == 0) ? x : gcd(y, x%y);
    }

    public static void main()
    {
        int m = System.in.read();
        int n = System.in.read();
        System.out.println(gcd(m,n));
    }
}
```