Programming Software Systems

Introduction to Programming for the Computer Engineering Track

Lecture 5 + Tutorial 5
An Introduction to Java

Eugene Zouev
Fall Semester 2020
Innopolis University

The Overall Structure of the Course

Three main parts of the course

- **The C language**

  Small, system-level (but still general-purpose) language

The Overall Structure of the Course

Three main parts of the course

The fall semester

- **The C language**
  Small, system-level (but still general-purpose) language
- **The Java language**
  Powerful application language

The Overall Structure of the Course

Three main parts of the course

The fall semester

- **The C language**

  Small, system-level (but still general-purpose) language

- **The Java language**

  Powerful application language

The spring semester

- **The C++ language**

  Fast and powerful general-purpose language **with deep semantics**

The Overall Structure of the Course

Three main parts of the course

The fall semester

- **The C language**

  Small, system-level (but still general-purpose) language

- **The Java language**

  Powerful application language

The spring semester

- **The C++ language**

  Fast and powerful general-purpose language **with deep semantics**

How Many Languages to Learn?

Important remark…

- There is NO "the best" programming language. Each language is good for its application domain and might be not that good for some other.

How Many Languages to Learn?

Important remark…

- There is NO "the best" programming language. Each language is good for its application domain and might be not that good for some other.
- Languages are very different *but…*
- Some basic principles behind them often are very much similar!

=> You shouldn't learn just a language – learn principles!
If so, you will be able to learn a new language in one week.

How Many Languages to Learn?

Important remark…

- There is NO "the best" programming language. Each language is good for its application domain and might be not that good for some other.
- Languages are very different *but…*
- Some basic principles behind them often are very much similar!

  => You shouldn't learn just a language – learn principles!
  If so, you will be able to learn a new language in one week.
- However, **the more languages you know the better** ☺.
  A software professional **must** know several programming languages.

How Many Languages to Learn?

There are many texts saying something like:

> "if you know the language X then it will be quite easy for you to learn our new wonderful language Y! – because we made it very similar to your favorite X!"

How Many Languages to Learn?

There are many texts saying something like:

> "if you know the language X then it will be quite easy for you to learn our new wonderful language Y! – because we made it very similar to your favorite X!"

Do not trust such statements!!!

The **syntax is the same** in all languages

```
class C { ... }
...
C c;
```

How Many Languages to Learn?

There are many texts saying something like:

> "if you know the language X then it will be quite easy for you to learn our new wonderful language Y! – because we made it very similar to your favorite X!"

Do not trust such statements!!!

The **syntax is the same** in all languages

```
class C { ... }
...
C c;
```

C++? – yes
Java? – yes
C#? – yes!

The same syntax & semantics in all languages

C++?
=========
Java?
C#?

Different semantics for C++ and for Java/C#

How Many Languages to Learn?

- This part of the course is based on the **Java language**.
  This doesn't mean it is *the best PL* over the world of programming…
- Later at Innopolis (and in your further professional career!) you will have to learn **many other PLs**. Be prepared and don't be afraid of it.

How Many Languages to Learn?

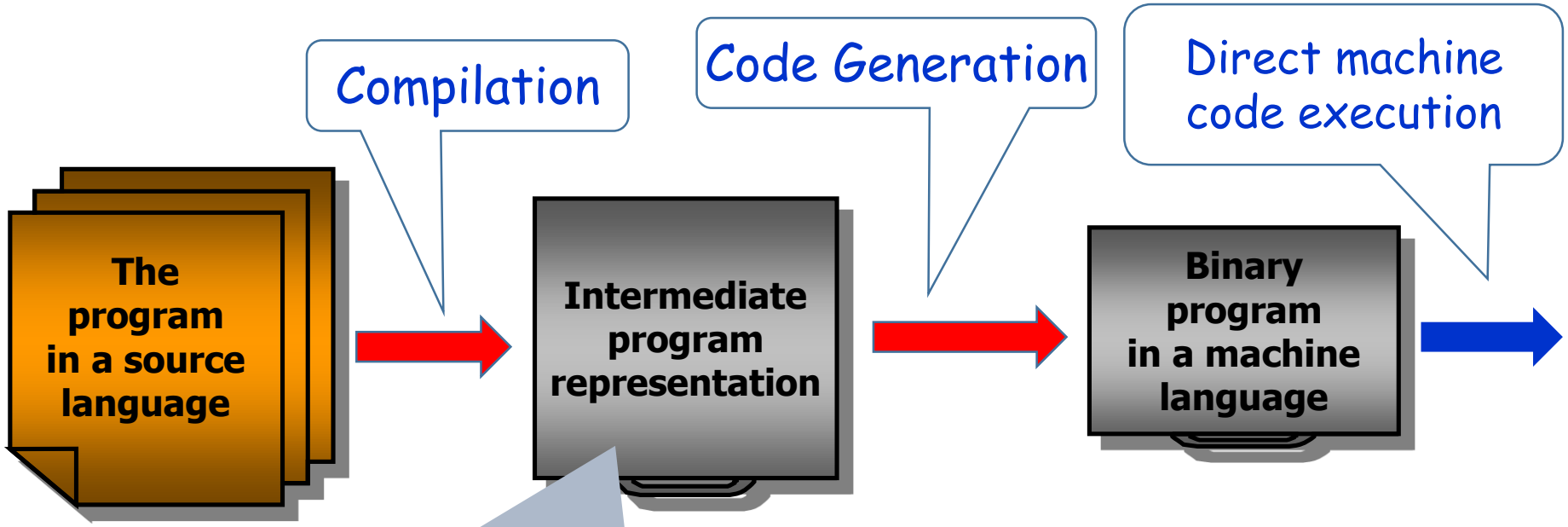
- This part of the course is based on the **Java language**.
  This doesn't mean it is *the best PL* over the world of programming…
- Later at Innopolis (and in your further professional career!) you will have to learn **many other PLs**. Be prepared and don't be afraid of it.
  - ***C#*** very similar to Java but is evolving much faster
  - ***C++*** the most powerful and complicated language ever
  - ***C*** middle-level language; "the (god)father" of many current PLs
  - **Javascript** for Web programming (is not similar to Java!!)
  - **Swift** Apple's alternative to Java/C#/C++/Objective C
  - **Go** Google's product for server-side programming
  - **Scala** Powerful extension of Java with functional programming
  - **Python** Dynamic language for scripting and application programming
  - **Eiffel** Systematically designed "contract based" OOP language
  - … *Many-many-many other languages*

The Plan for Today

- Program compilation & execution:
  - Conventional model
  - The Java model
- Memory model: code, stack & heap
- The structure of Java programs
- A gentle introduction to OOP
- The notion of class (without OOP ☺)

Compilation & Execution: Conventional Model

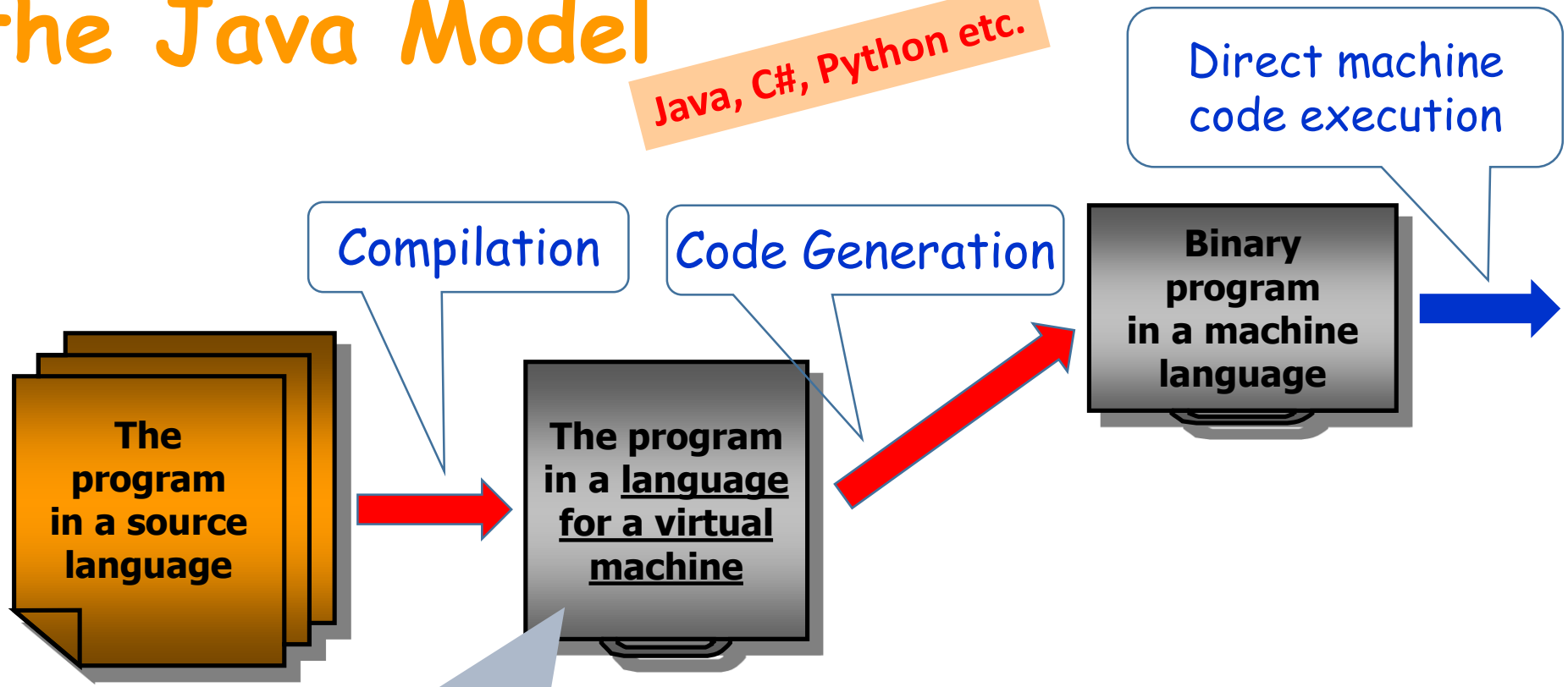
Pascal, C/C++, Eiffel, Go etc.



- Regular representation; only for internal purposes
- Low-level and semantically poor representation
- Targeted mainly to generate machine code for real hardware

Compilation & Execution: the Java Model

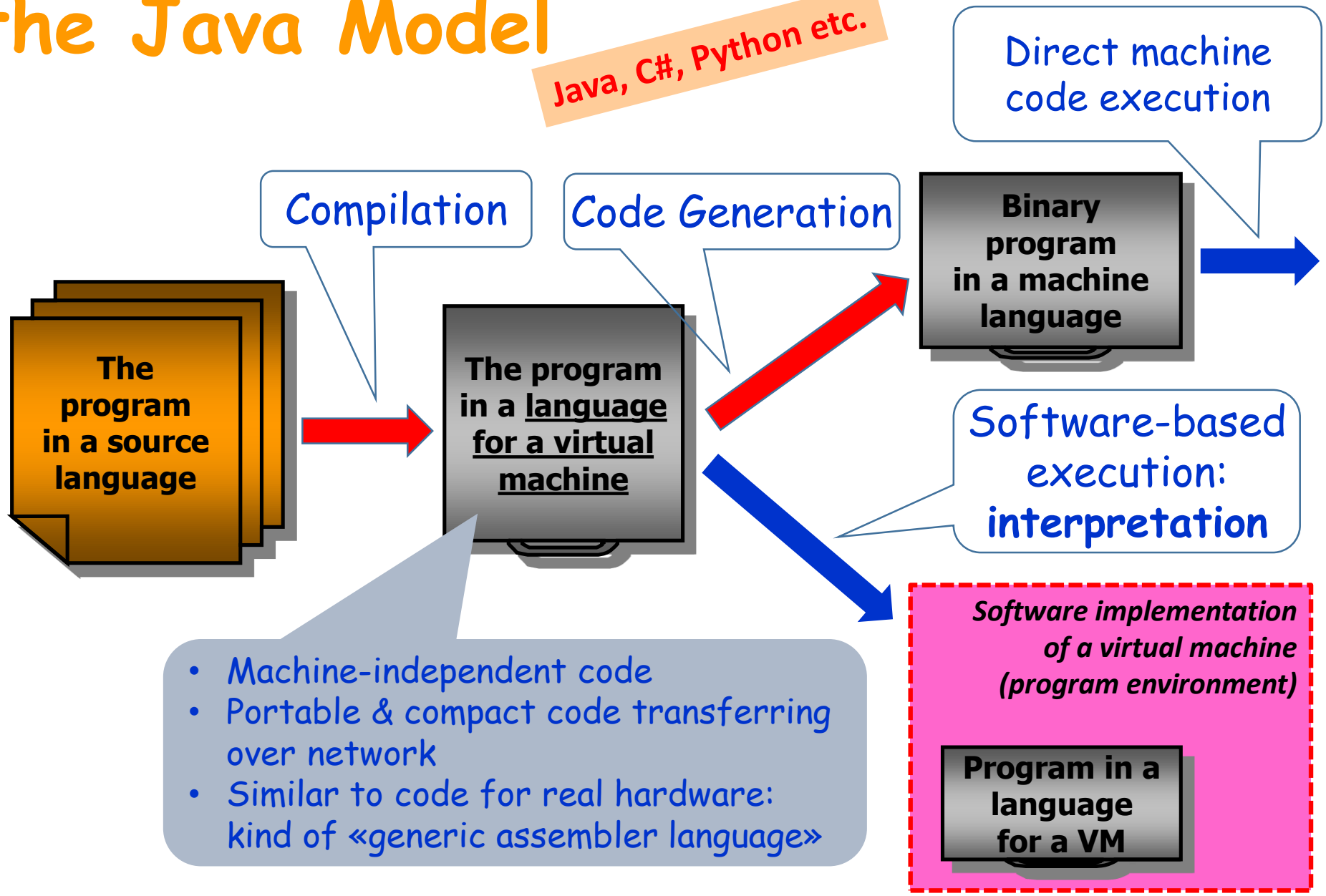
Java, C#, Python etc.



- Machine-independent code
- Portable & compact code transferring over network
- Similar to code for real hardware: kind of «generic assembler language»

Compilation & Execution: the Java Model

Java, C#, Python etc.



- Machine-independent code
- Portable & compact code transferring over network
- Similar to code for real hardware: kind of «generic assembler language»

- Interpretation
- Just-in-time compilation (JIT)
- Ahead-of-time compilation (AOT)

Java Virtual Machine (JVM)

- Java programs get compiled not to machine code for a particular hardware architecture, but to code for some **hypothetical** (**abstract**, **virtual**) computer.
- This "computer" (actually, system software) has all architectural features of a real computer: a "CPU" with instruction set, with memory, registers etc.
- Such a "computer" is called **virtual machine**. Instructions being executed by this virtual machine are called **bytecode**.

Java Slogan:

Write once – run everywhere

Java Virtual Machine (JVM)

- Java programs get compiled not to machine code for a particular hardware architecture, but to code for some **hypothetical** (**abstract**, **virtual**) computer.
- This "computer" (actually, system software) has all architectural features of a real computer: a "CPU" with instruction set, with memory, registers etc.
- Such a "computer" is called **virtual machine**. Instructions being executed by this virtual machine are called **bytecode**.

Java Slogan:

Java Virtual Machine: Major Features

- Hardware independence
  - however, rather "close" to real machines
- **Stack-based execution model**
  - not only function calls, but expression calculations as well
- Rather high level of the instruction set
  - high-level function call mechanism; exception mechanism is supported; rather compact code
- Advanced code structure;
  - constants, metadata (!), debug information
- Open format:
  - complete & detailed documentation

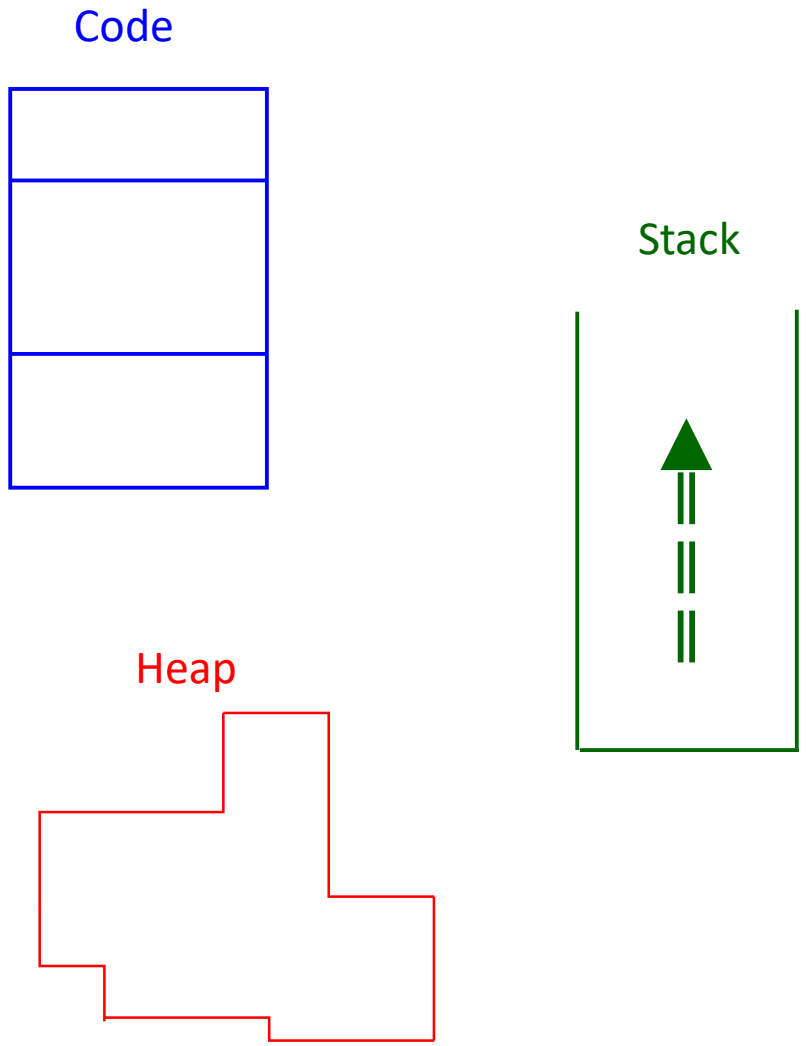
Memory Model

There are different models of the program execution. In the model we use, when started, program is assigned **three separate and independent portions of memory** (referred to as address space of a running program - aka process). These are:

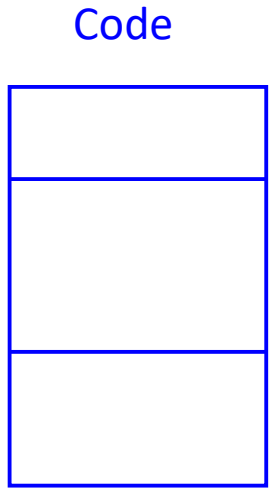
- **Code** area
  - where code to be executed is loaded & stored
- **Stack** (or **execution stack**)
  - used to perform computation,
  - store local variables and
  - perform function call management
- **Heap**, or dynamic memory area
  - used to store variables and objects allocated dynamically

Memory Model

- The code and the heap area can be accessed with no special restrictions
- The stack area is accessed using a **LIFO (Last In – First Out)** policy
- The most languages are "stack-based"

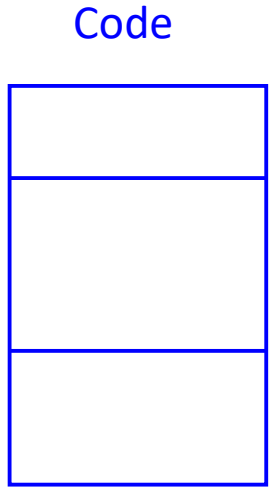


Memory Model & Management: Code



- ***C, C++***
  - We can assume that the **entire code** is loaded in the code area (neglecting issues like dynamic linking…)
  - **stack** is managed by the hardware (if possible)
  - **heap** is managed by run-time support software (OS)

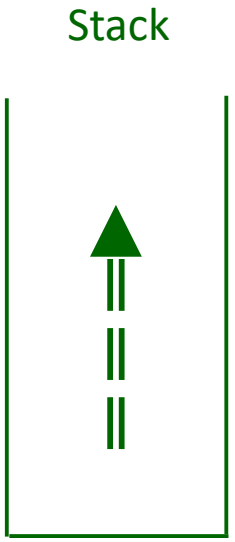
Memory Model & Management: Code



- ***C, C++***
  - We can assume that the **entire code** is loaded in the code area (neglecting issues like dynamic linking…)
  - **stack** is managed by the hardware (if possible)
  - **heap** is managed by run-time support software (OS)
- **Java**
  - code is loaded on the **class-by-class** basis: the execution (actually, the JVM) proceeds loading and running new classes when the need arises, according to the flow of the computation

    The first class to be loaded is a special class containing the static "main" method (will consider it later)
  - **stack** & **heap** are managed by the JVM

Memory Model: Stack



- In most modern languages (including Java) the execution is centered around the **execution stack**.
- All algorithms are organized into functions (sometimes called procedures, methods, routines etc.)
- The order of execution of functions is LIFO, i.e. the last function called is the first to terminate (this behavior is obtained using the stack)

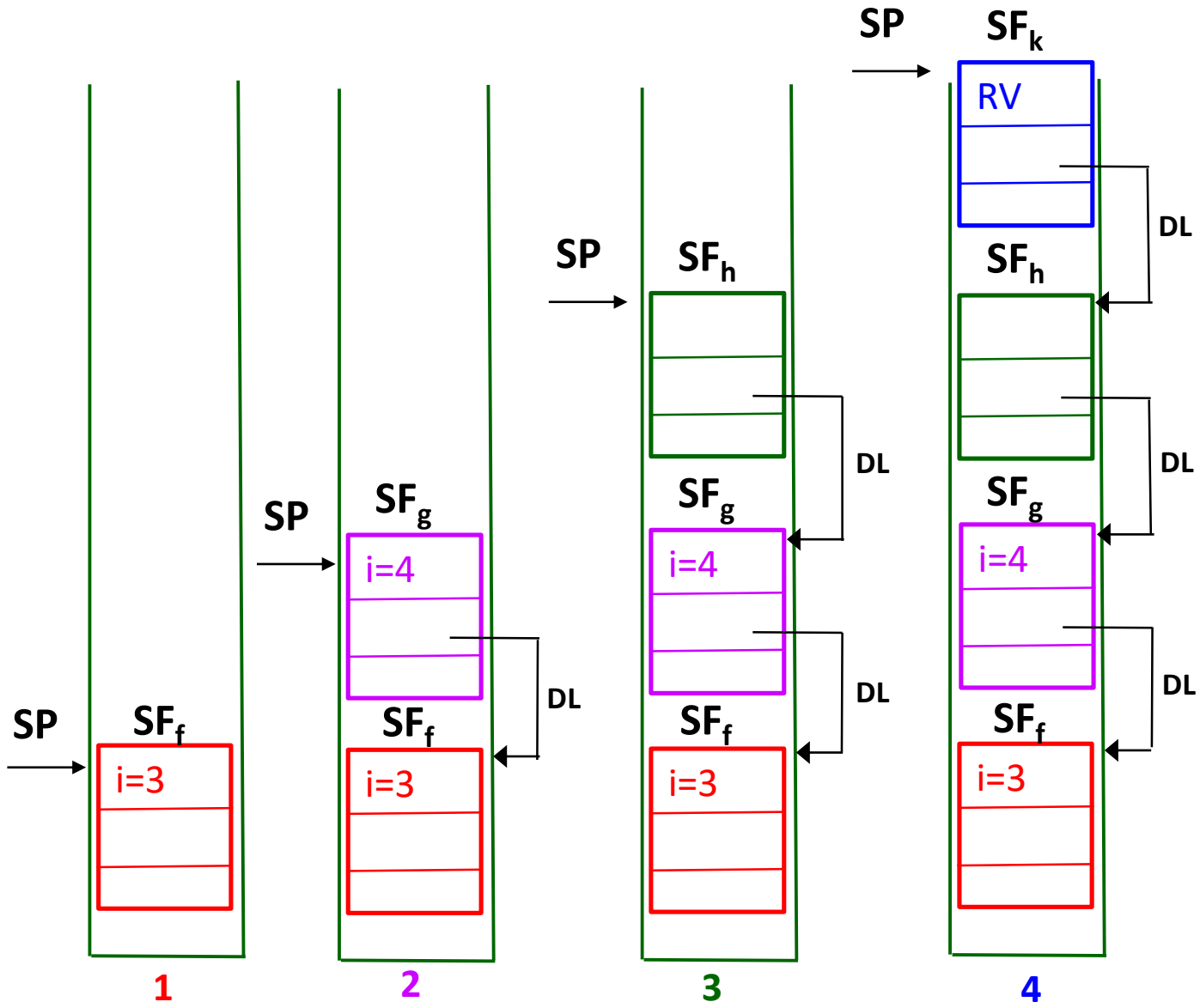
Typical operations on stack:
push, pop, empty

Memory Model: Stack for Function Calls

Call chain

```
void f() {
    int i = 3;
    g();
}
void g() {
    int i = 4;
    i = i + h();
}
int h() {
    return 1 + k(5);
}
int k(int z) {
    return z+1;
}
```

Return chain



Memory Model: Stack for Function Calls

- Each time a function is called, **all the information specifically needed for the function** execution are put on the stack (function call arguments, local variables, reference to the information about the previously called function etc.).
- That information is collectively called the **stackframe** of the function call.
- This allows recursion, since for each call there will be a separate activation record on the stack.
- When the call is completed (the function "returns") the corresponding AR is destroyed ("popped out" of the stack).
- Activation records are organized from bottom to top in memory diagram (see the prev. slide).

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**

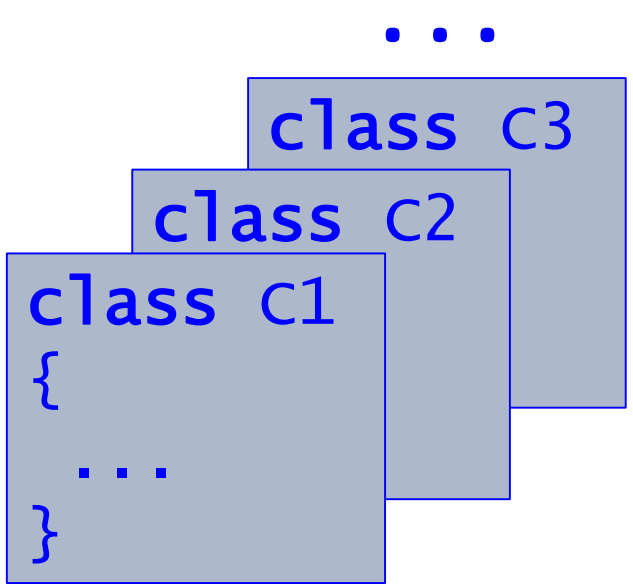
The Structure of Java Programs

- 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!

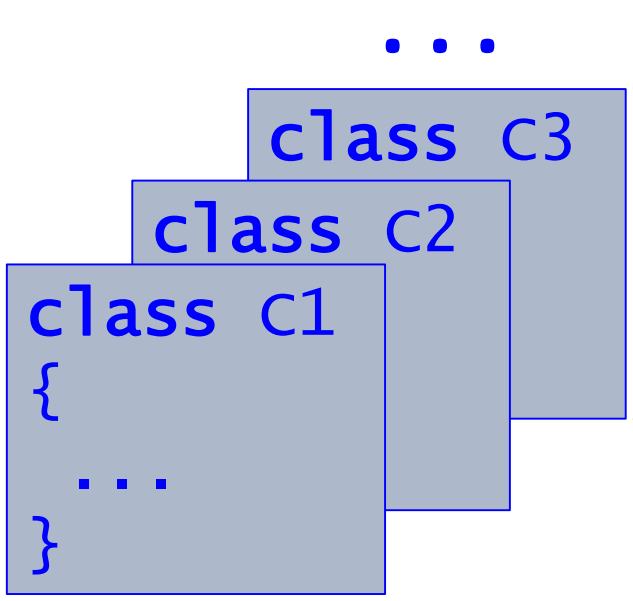
The Structure of Java Programs



Program:
collection of classes

Simplified!

The Structure of Java Programs



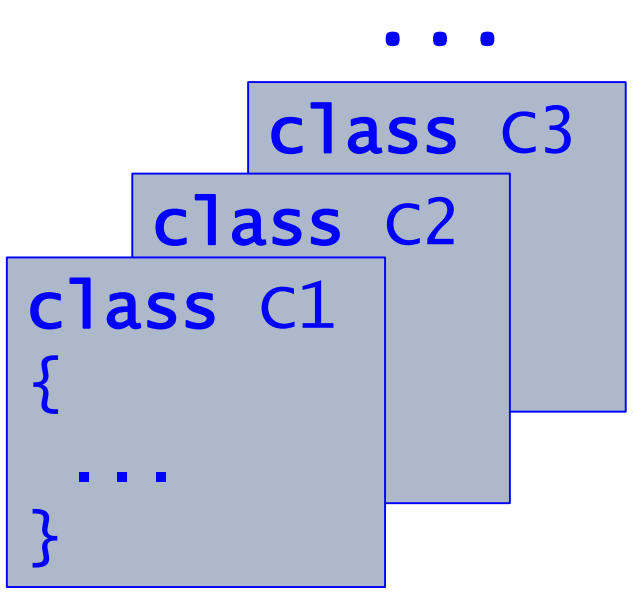
Program:
collection of classes

```
class C3
{
  int a, b, c;
  ...

  void f1()
  { working with a }

  int f2()
  { working with b, c }
    ...
}
```

Class:
collection of data & functions

Simplified!

The Structure of Java Programs



Program:
collection of classes

```
class C3
{
  int a, b, c;
  ...

  void f1()
  { working with a }

  int f2()
  { working with b, c }
    ...
}
```

Class:
collection of data & functions

Simplified!

For comparison:
How it is in *C*

```
struct C3
{
   int a, b, c;
};
 ...

void f1(C3* c)
{ ... }

int f2(C3* c)
{ ... }
```

OOP: Origins & Present

Simula-67 (based on Algol-60):

The concept of **class**

Smalltalk: "Pure" OOP: its features are centered around the notions of class and object

Classes & **message passing**

C++: "Hybrid" language: it mixes typical OO features with those typical to classic imperative, stack-based languages (like C, Ada, Pascal)

OOP as industry-level technique

Java: "Pure" OOP: its features are centered around the notions of class and object

OOP worldwide

Object-Oriented Approach

Basic idea

- A computer program is a **model** (reflection, representation) of a (part of the) **real world**, or of an application domain.

Object-Oriented Approach

Basic idea

- A computer program is a **model** (reflection, representation) of a (part of the) **real world**, or of an application domain.
- The real world consists of a set of related and communicating **objects**.
- Therefore, to create an adequate model of the real world we need a means that would reflect/represent objects of the world, and their relationships.

Object-Oriented Approach

Basic idea

- A computer program is a **model** (reflection, representation) of a (part of the) **real world**, or of an application domain.
- The real world consists of a set of related and communicating **objects**.
- Therefore, to create an adequate model of the real world we need a means that would reflect/represent objects of the world, and their relationships.
- There are a lot of various and different **kinds of objects** in the real world; the set of objects is (potentially) **infinite**.
- Therefore, we need a mechanism that could **uniformly** represent the structure and behavior of the all objects.

What is Class?

Bjarne Straustrup:

Класс непосредственно выражает некое понятие в программе. Класс – это тип, определённый пользователем. Он определяет, как представляются объекты этого класса, как они создаются, используются и уничтожается.

Если вы размышляете о чём-то как об отдельной сущности, то вполне возможно, должны определить класс, представляющий эту «вещь» в программе.

Примерами [*классов*] служат вектор, матрица, поток ввода, строка, быстрое преобразование Фурье, **клапанный регулятор**, **рука робота**, драйвер устройства, рисунок на экране, диалоговое окно, график, окно, **термометр**, **часы**.

Programming Principles and Practice Using C++

Classes & Objects

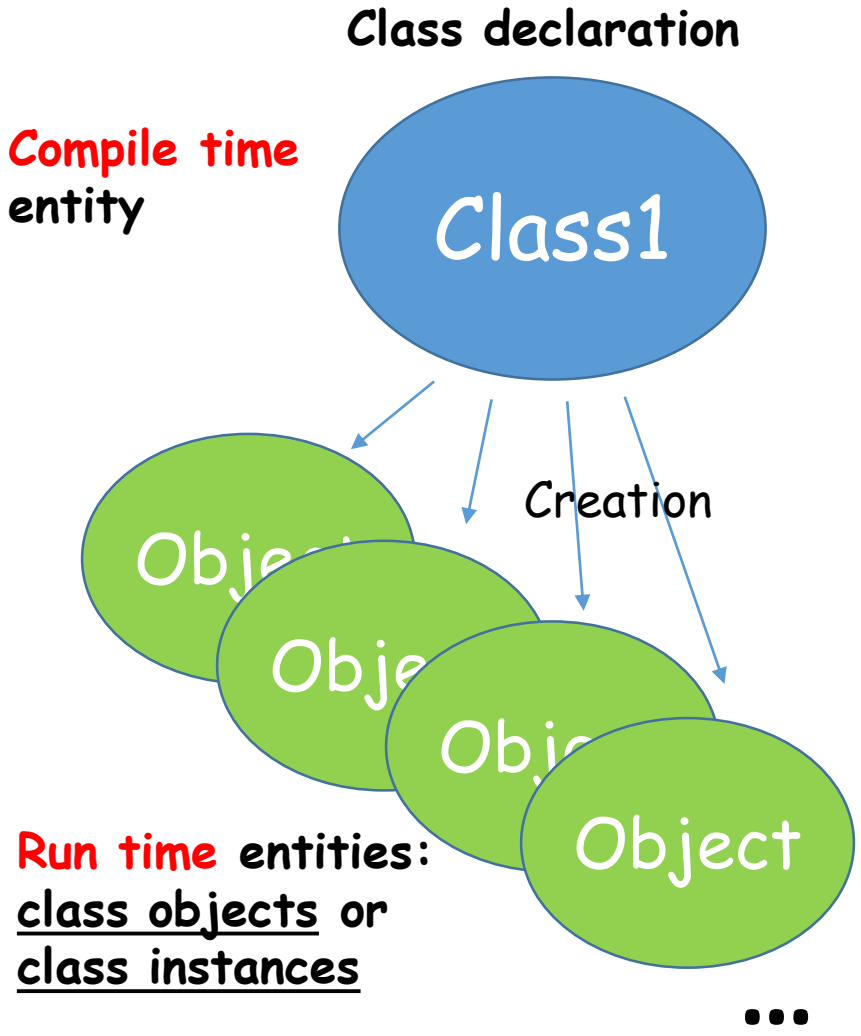
Class declaration

Compile time
entity

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

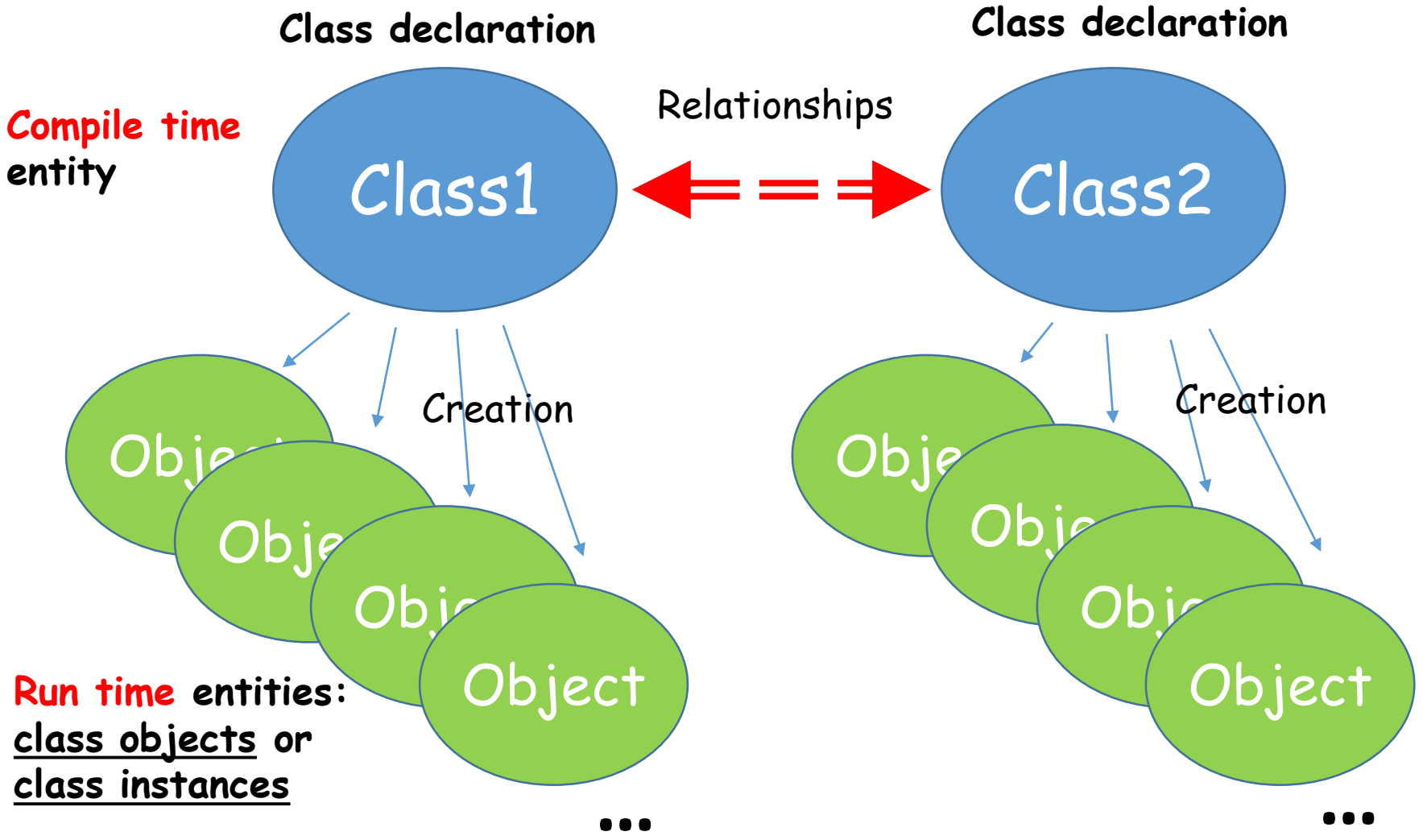
Classes & Objects

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



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.



Classes & Objects

- A class represents the general properties as well as the structure shared by a group of entities: the **instances** (or **objects**) of that class.
- An object is **an instance of a class**, a class *in action*.
- Object is a particular entity, which shares the general structure and behavior with all the other instances of the class it belongs to.
- Conceptual relationship between classes and objects is similar to that which exists between an **abstract idea** and a **concrete example of it**.
  - e.g. the idea of dog and a particular dog Rex
  - e.g. the idea of a point and real points composing a picture
  - e.g. the idea of a vehicle and various car models and concrete cars

Classes & Objects

- A class represents the general properties as well as the structure shared by a group of entities: the **instances** (or **objects**) of that class.
- An object is **an instance of a class**, a class *in action*.
- Object is a particular entity, which shares the general structure and behavior with all the other instances of the class it belongs to.

Classes & Objects

- A class represents the general properties as well as the structure shared by a group of entities: the **instances** (or **objects**) of that class.
- An object is **an instance of a class**, a class *in action*.
- Object is a particular entity, which shares the general structure and behavior with all the other instances of the class it belongs to.
- Conceptual relationship between classes and objects is similar to that which exists between an **abstract idea** and a **concrete example of it**.
  - e.g. the idea of dog and a particular dog Rex
  - e.g. the idea of a point and real points composing a picture
  - e.g. the idea of a vehicle and various car models and concrete cars

Role of Classes in Software Development

- Classes are **units of data abstraction**
  - Each class should, ideally, define a unique and cohesive behavior, with limited coupling to other classes
- Classes are **units of interaction**
  - In "pure" object oriented languages, the task of a program is accomplished by objects (instances of classes) interacting among themselves
- Classes are **units of development**
  - Classes are assigned to programmers for development and to testers to test
- Classes and objects may also be used as **units of requirement and planning**
  - Agile methods develop user stories with interacting objects to negotiate with the customers the work to do
  - new planning techniques use objects counts to estimate effort

The First Class Example

- So.. I want to create a program drawing figures on the screen.
- I know that figures are to be composed from lines/curves – and the latter consist of points.
- Therefore, the notion of **point** will be the basic notion in my program.

```
class Point
{




}
```

The First Class Example

- So.. I want to create a program drawing figures on the screen.
- I know that figures are to be composed from lines/curves – and the latter consist of points.
- Therefore, the notion of **point** will be the basic notion in my program.
- Each point is characterized, first of all, by its coordinates, say, x and y. These coordinates are primary **features** or **properties**, or **attributes** of each point.

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




}
```

The First Class Example

- So.. I want to create a program drawing figures on the screen.
- I know that figures are to be composed from lines/curves – and the latter consist of points.
- Therefore, the notion of **point** will be the basic notion in my program.
- Each point is characterized, first of all, by its coordinates, say, x and y. These coordinates are primary **features** or **properties**, or **attributes** of each point.
- Also, I would like to perform some **operations** on points: for example, to move it to a new position.
- …

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

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

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;
    }
}
```

Objects

To conclude:

- Objects are created (occur) & destroyed (disappear) **dynamically** while program execution.
- Objects have a **state**, i.e., have data stored in memory. The state "structure" is the same but values can be different.
- Objects have a "**behavior**", i.e., they are "machines" offering operations (features, methods). Different objects always have **the same** set of operations.
- Objects are in some **relations** with other objects of the same class or with objects of some other class(es).

For example, objects can compose **arrays**, or can be parts of other objects – will see later

Objects: How to Create

- Creation (instantiating) an object means creating an object of a given class.
- There is a special operator in Java for creating objects: **new**.

```
new Point()
```

Objects: How to Create

- Creation (instantiating) an object means creating an object of a given class.
- There is a special operator in Java for creating objects: **new**.

```
new Point()
```

Syntactically, this is **unary operator** like –x or +a where the **new** keyword plays the role of the operator sign, and the name of the class is its "operand".

Objects: How to Create

- Creation (instantiating) an object means creating an object of a given class.
- There is a special operator in Java for creating objects: **new**.

```
new Point()
```

Syntactically, this is **unary operator** like −x or +a where the **new** keyword plays the role of the operator sign, and the name of the class is its "operand".

What's **semantics** of the **new** operator?

- **Memory is allocated** to keep the current state of the new object (for x and y in our example).
- The memory is allocated **in the heap**.
- State of the object **is initialized** by some default values.
- The result of the operator is the **reference** to the object.

Objects and References

```
class Point
{
    int x, y;
    void move(int dx, int dy)
    {
        x += dx;
        y += dy;
    }
}
...
class OtherClass
{
    void f()
    {
        Point p = new Point();
        ...
    }
}
```

Objects and References

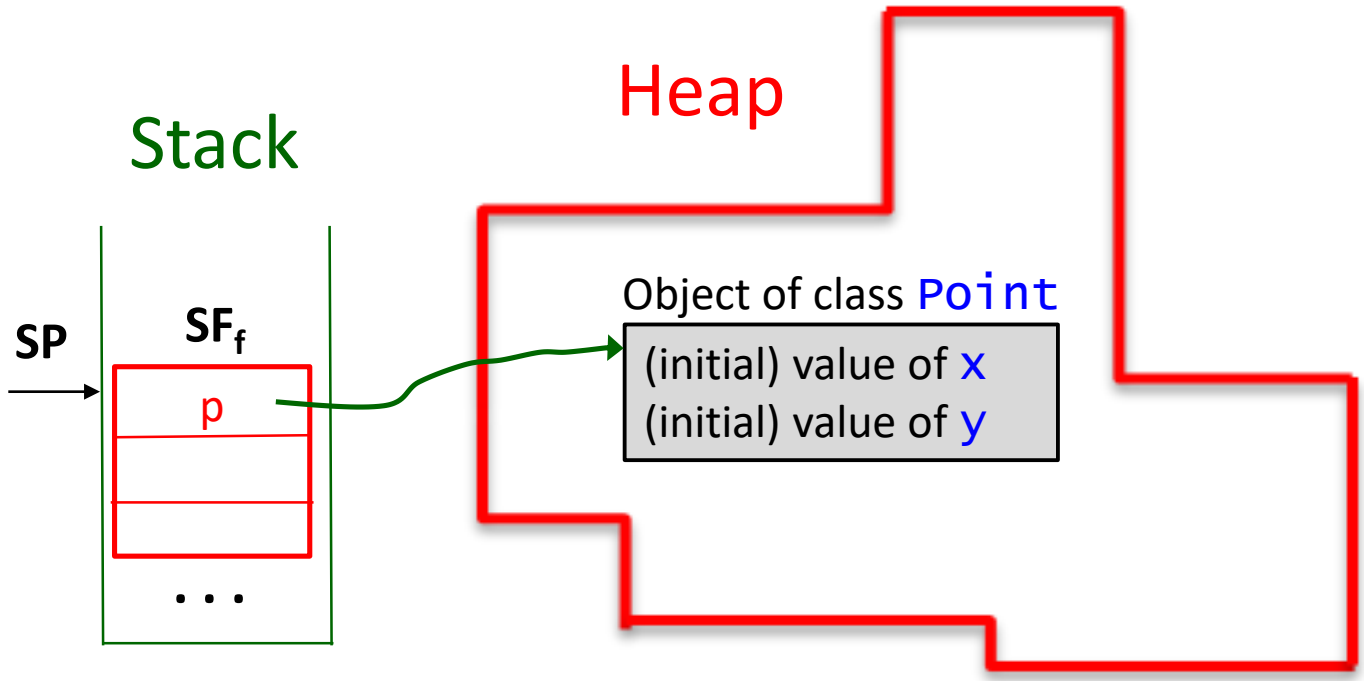
```
class Point
{
    int x, y;
    void move(int dx, int dy)
    {
        x += dx;
        y += dy;
    }
}
...
class OtherClass
{
    void f()
    {
        Point p = new Point();
        ...
    }
}
```

- The object just created by **new** **doesn't have a name**.
- In order to use it we have to **assign** the result of **new** to an object of type `Point`.
- Now we can work with the new object **by using the reference to it**.

Objects and References

```
class Point
{
    int x, y;
    void move(int dx, int dy)
    {
        x += dx;
        y += dy;
    }
}
...
class OtherClass
{
    void f()
    {
        Point p = new Point();
        ...
    }
}
```

- The object just created by `new` **doesn't have a name**.
- In order to use it we have to **assign** the result of `new` to an object of type `Point`.
- Now we can work with the new object **by using the reference to it**.

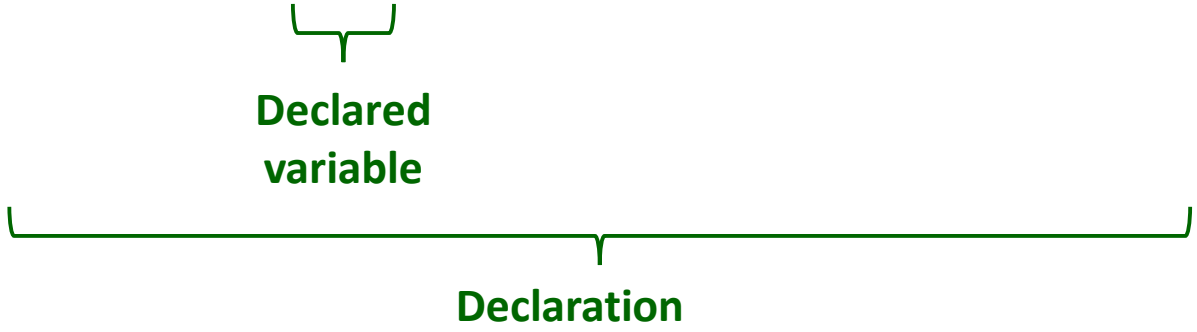


Objects and References

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

Objects and References

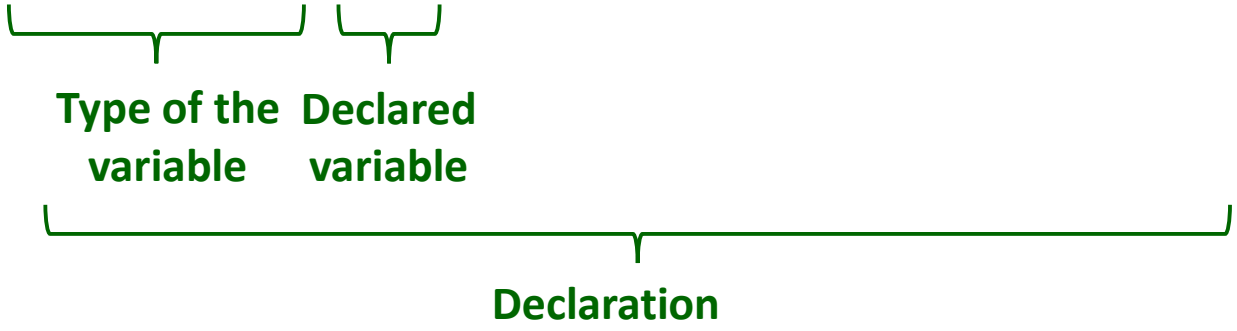
```
Point p = new Point();
```



- This is the **declaration**. The variable p is declared.

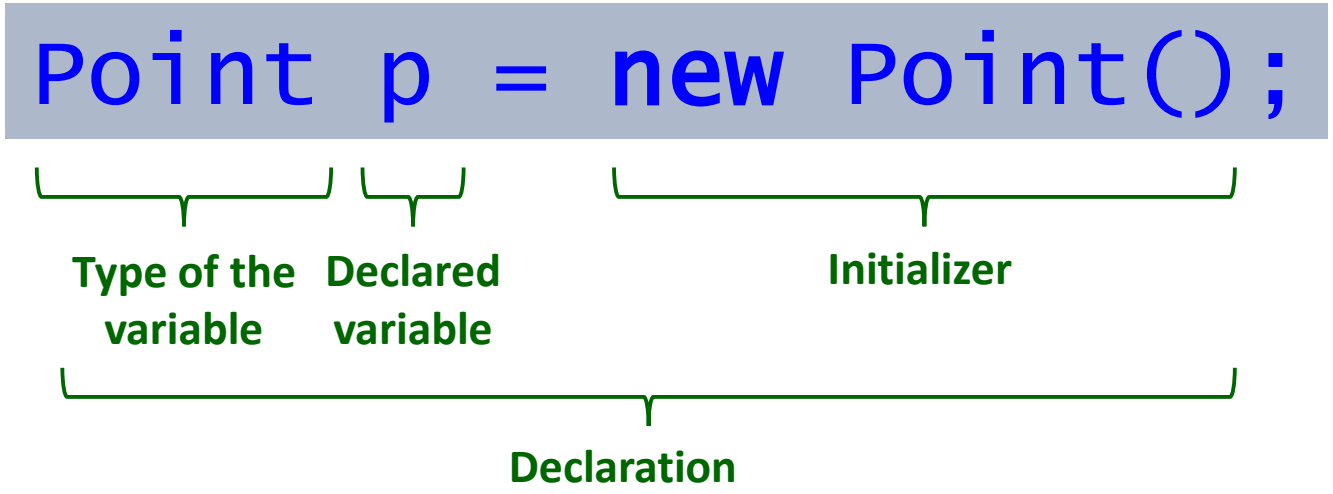
Objects and References

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



- This is the **declaration**. The variable `p` is declared.
- The type of `p` is class type `Point`. This means that `p` can refer to an object of type `Point`.

Objects and References



- This is the **declaration**. The variable `p` is declared.
- The type of `p` is class type `Point`. This means that `p` can refer to an object of type `Point`.
- The declaration contains **initialization**. This means that the initial value of `p` is the value of the expression after `=`. The value of the expression is the result of the unary **`new`** operator, i.e., the reference to the object of class `Point` just created.

Value and Reference Types

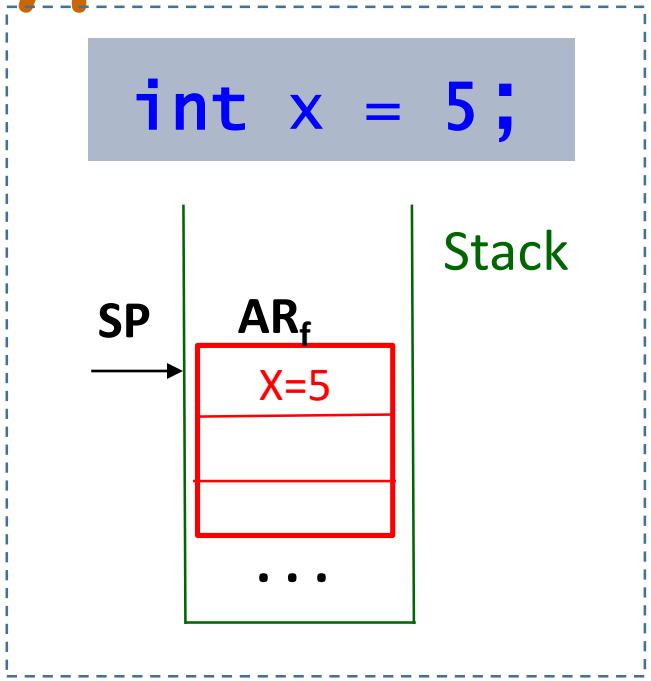
- There are two categories of types in Java: **value types** and **reference types**.

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

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:

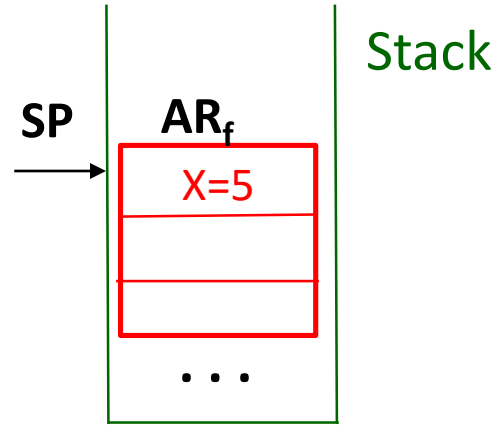


Value and Reference Types

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

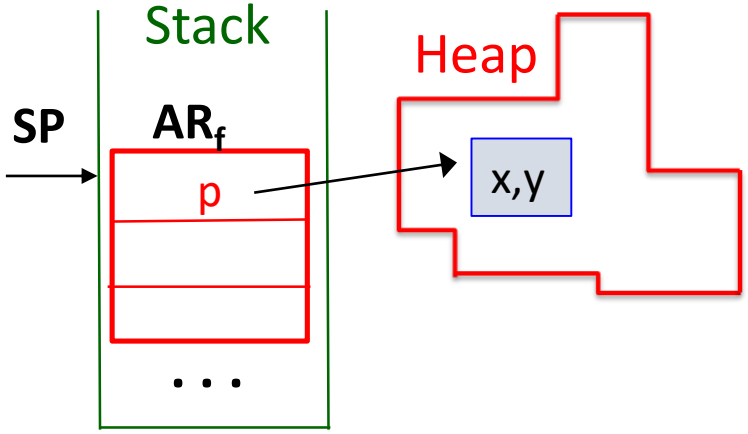
  Examples of value types: integers, floating, doubles. Values of these types are represented directly:

```
int x = 5;
```



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

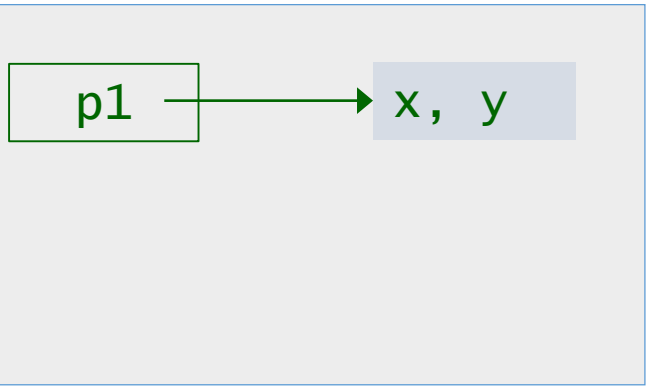
```
Point p = new Point();
```



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

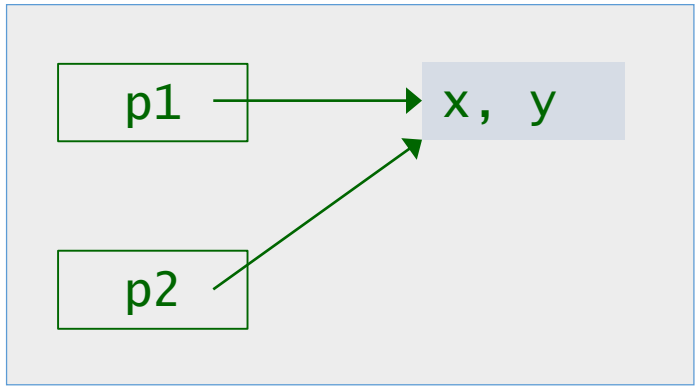
Objects and References

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

class OtherClass
{
    void f() {
        Point p1 = new Point;
        Point p2 = p1;
        {
            Point p3 = new Point;
        }
        ...
    }
}
```



Objects and References

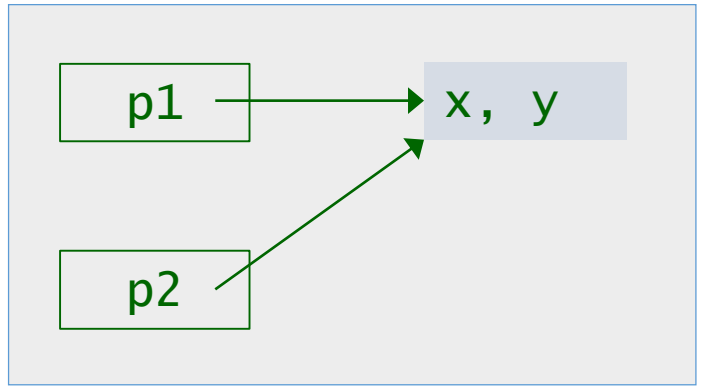
```
class Point
{
    int x, y;
    void move(int dx, int dy)
    {
        x += dx;
        y += dy;
    }
}

class OtherClass
{
    void f() {
        Point p1 = new Point;
        Point p2 = p1;
        {
            Point p3 = new Point;
        }
        ...
    }
}
```



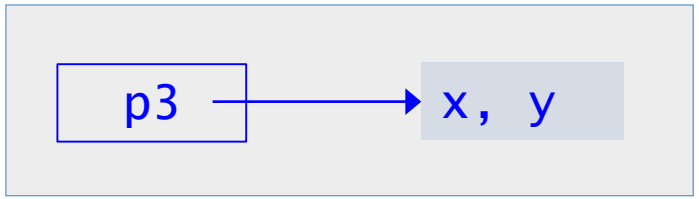
Two references refer to (share) the same instance

Objects and References

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

class OtherClass
{
    void f() {
        Point p1 = new Point;
        Point p2 = p1;
        {
            Point p3 = new Point;
        }
        ...
    }
}
```

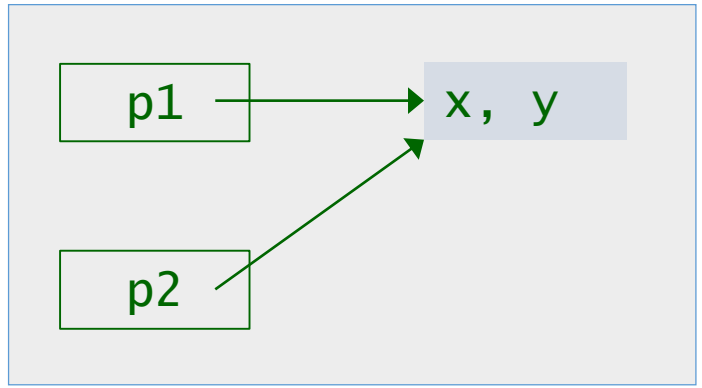


Two references refer to (share) the same instance

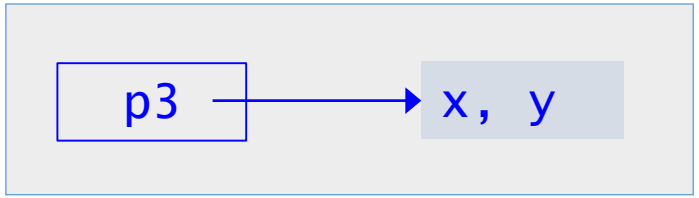


Objects and References

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

class OtherClass
{
    void f() {
        Point p1 = new Point;
        Point p2 = p1;
        {
            Point p3 = new Point;
        }
        ...
    }
}
```



Two references refer to (share) the same instance



The instance still exists but there is no way to access it: The instance is lost!

Access to Class Instances

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

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

So… we have declared the class, we know how to create instances of the class…

But what can we **do with instances** after creating them? ☺

Access to Class Instances

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

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

```
Point p = new Point();
...
p.move(1,3);
```

So… we have declared the class, we know how to create instances of the class…

But what can we **do with instances** after creating them? ☺

The common rule is that that **the only way** to access instance's features (members) is to use a reference to the instance. The whole construct is called **dot notation**.

Access to Class Instances

Dot notation, the common form:

Do you remember about the similar construct in C?

```
ref_to_instance . member_name
```

But this is not enough to know about access… ☺

Access to Class Instances

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

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

By default, all class members are **not accessible** (i.e., **private**).

Access to Class Instances

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

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

By default, all class members are **not accessible** (i.e., **private**).

To make them accessible (**public**) you should mark them public **explicitly**.

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

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

Private (inaccessible) by default

Public (accessible) by explicit specification

Access to Class Instances

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

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

By default, all class members are **not accessible** (i.e., **private**).

To make them accessible (**public**) you should mark them public **explicitly**.

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

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

Private (inaccessible) by default

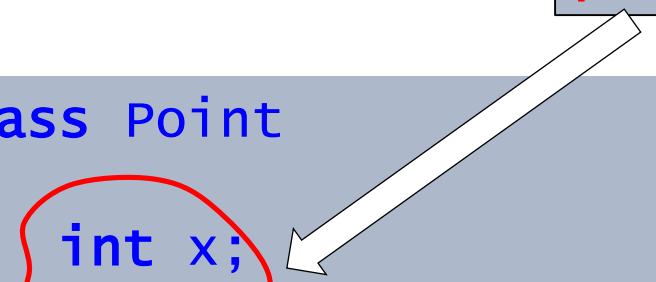
Public (accessible) by explicit specification

```
Point p = new Point();
...
p.move(1,3); // OK
p.x = 7; // Error
```

Interface & Implementation

Can be specified explicitly:

```
private int x;
private int y;
```



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

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

Interface & Implementation

Class **implementation**:

How class instances are organized internally. This is **hidden** from clients of the class.

Can be specified explicitly:

```
private int x;
private int y;
```

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

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

Interface & Implementation

Class **implementation**:

How class instances are organized internally. This is **hidden** from clients of the class.

Class **interface**:

How class communicates with its clients, OR:

How clients accept (understand, work with) class instances.

This is **accessible** for clients of the class.

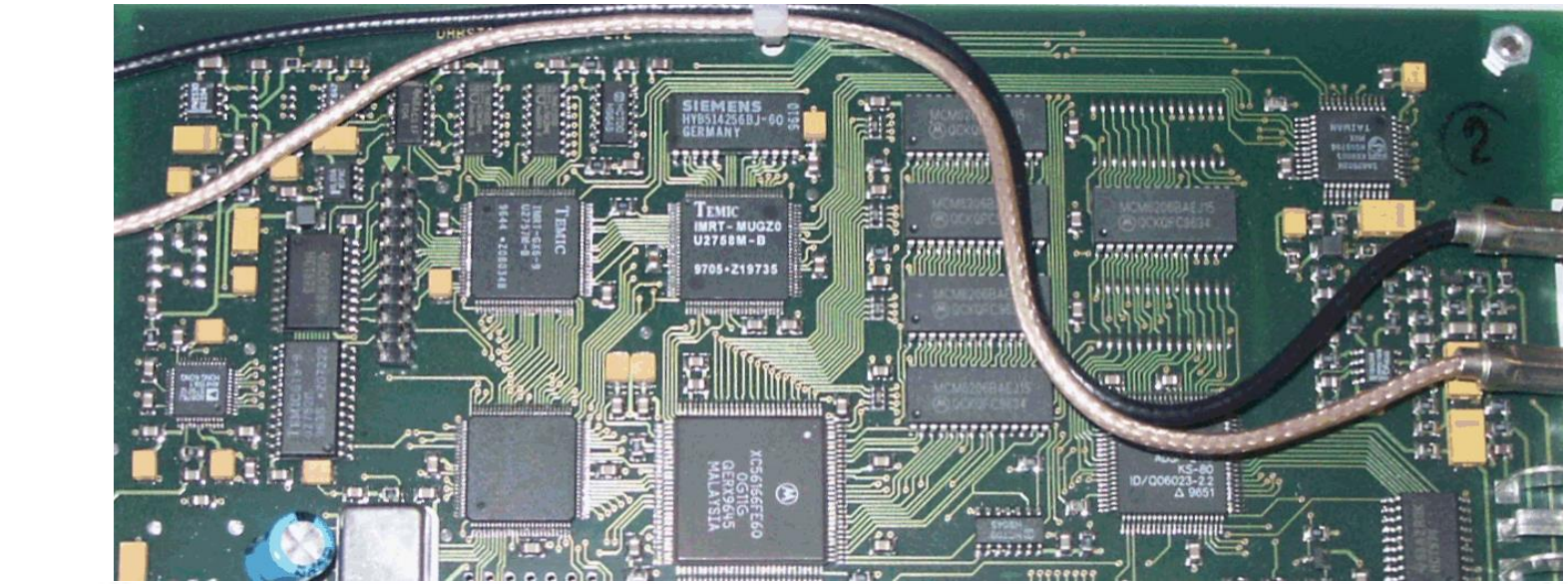
Can be specified explicitly:

```
private int x;
private int y;
```

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

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

Interface & Implementation

Interface & Implementation



Interface & Implementation

Interface:
what a client sees

Interface & Implementation

Interface:
what a client sees

Implementation:
what's under the hood

Constructors

OR: How to initialize class instances

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

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

```
class SomeOtherClass
{
    Point p = new Point();
    ...
}
```

The problem: what are (initial) values of x & y?

Constructors

OR: How to initialize class instances

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

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

```
class SomeOtherClass
{
    Point p = new Point();
    ...
}
```

The problem: what are (initial) values of x & y?

The solution: **constructor**

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.

```
class Point
{
  int x, y;

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



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

```
class SomeOtherClass
{
  Point p1 = new Point();

}
```

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 Point
{
  int x, y;

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

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

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

Destroying Class Instances

What to do when you don’t need an instance anymore?
Or: how to deal with instances with lost references?

- .

Destroying Class Instances

What to do when you don't need an instance anymore?
Or: how to deal with instances with lost references?

- The answer is: **nothing** to do.

Java uses **automatic garbage collection**; the programmer does not have to deal with memory management.

- Objects with no references pointing to them are considered eligible for automatic garbage collection by the system
- The garbage collector runs periodically and performs the real destruction of these objects.
- Thus explicit object destruction is (almost) never an issue in Java (except in JNI and connection to database).
- Garbage collection is not directly under control of the programmer, hence problems could arise if strictly predictable timing behavior is needed (as in real-time systems).

Conclusion

What we have learnt today:

- **Object-oriented approach** to programming: basic idea (to be discussed in more details later these days)
- **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**