Programming Software Systems

Introduction to Programming for the Computer Engineering Track

Lecture 7 + Tutorial 7
An Introduction to Java

Eugene Zouev
Fall Semester 2020
Innopolis University

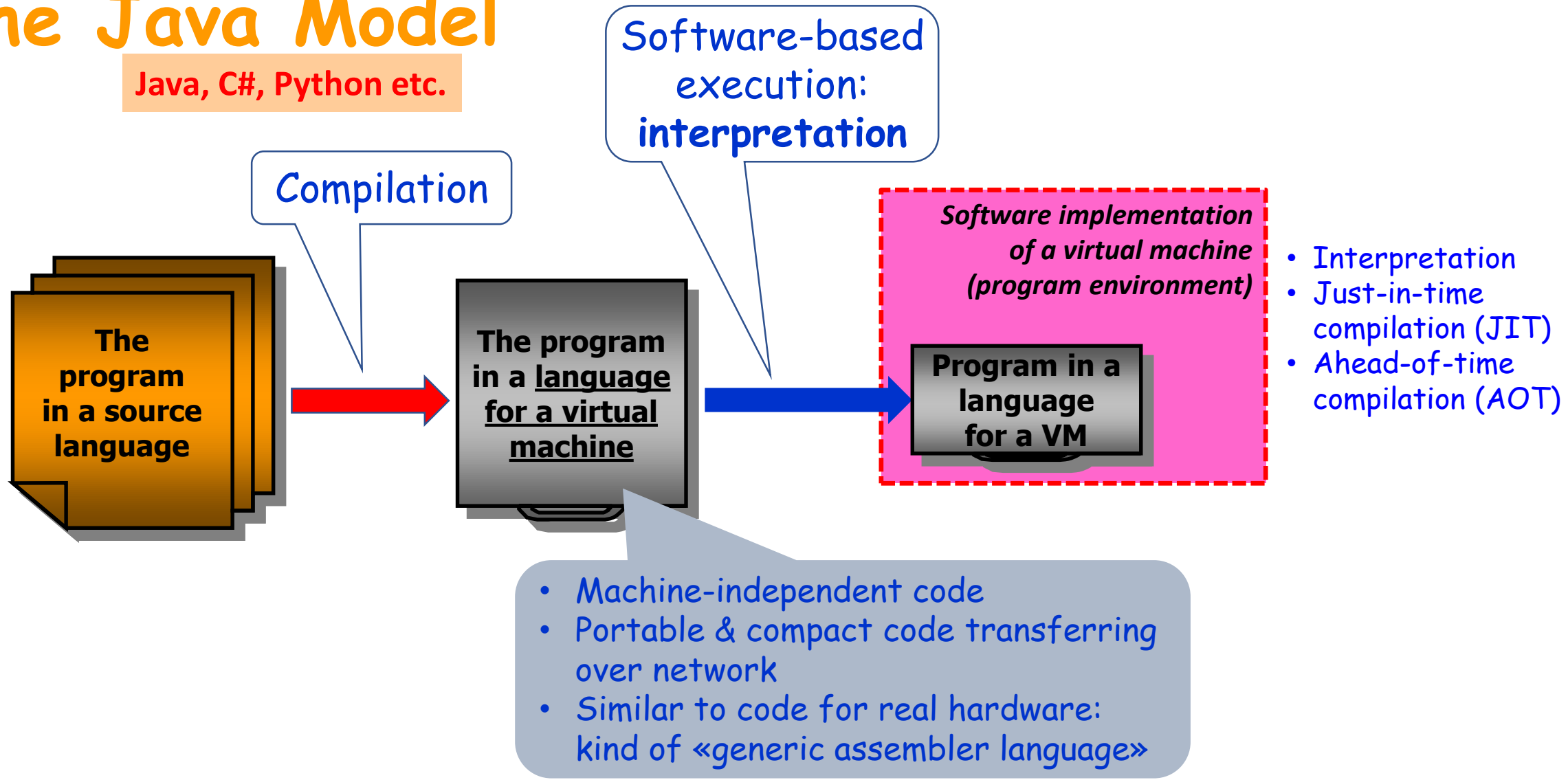
What We Have Learnt

- **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**
- `null` & `this`

Compilation & Execution: the Java Model

From the previous lecture

Java, C#, Python etc.



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

The Structure of Java Programs

From the previous lecture

- 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:

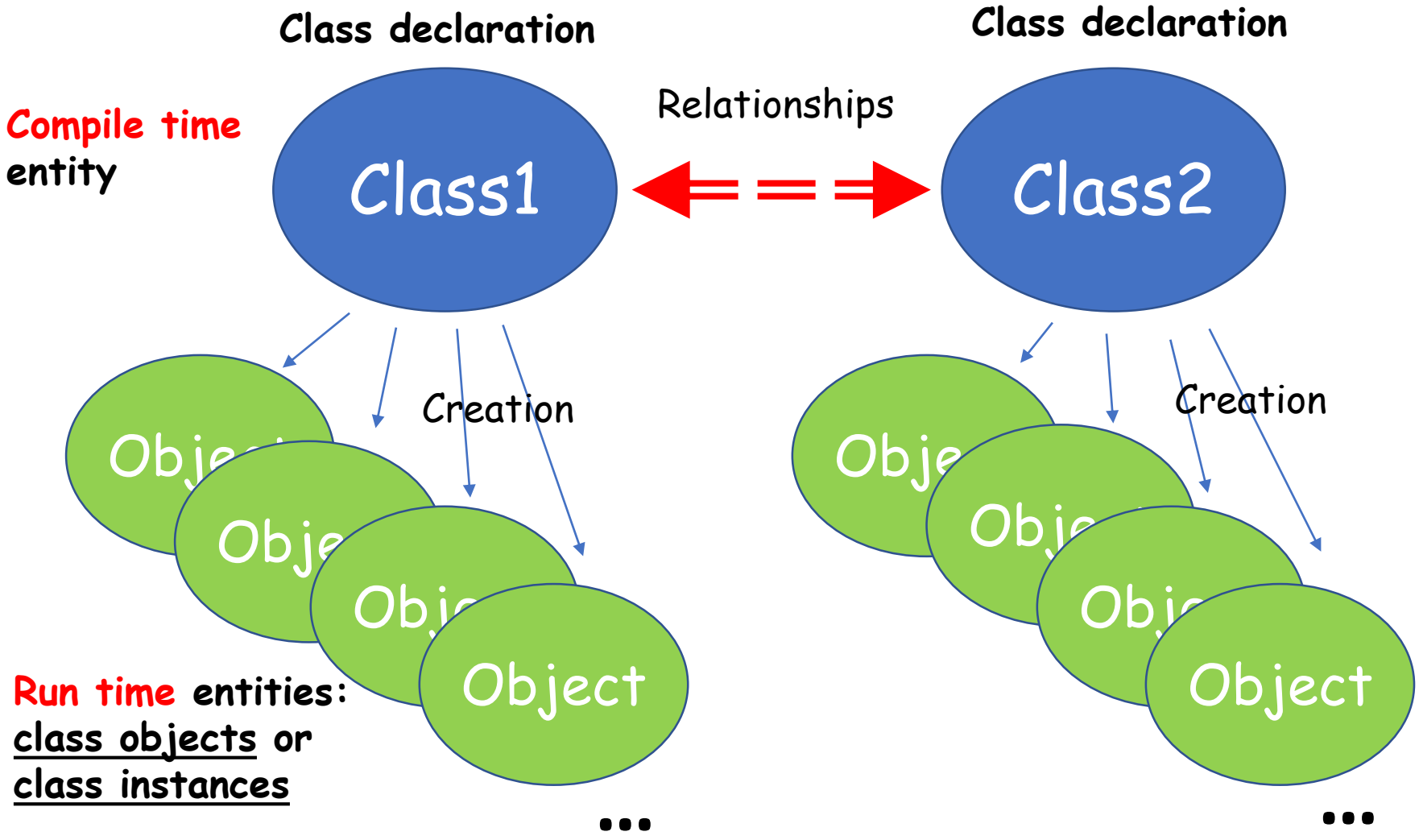
Class is a language construct comprising **algorithms** (in form of **functions**) and **data** the algorithms work on

Simplified!

Classes & Objects

From the previous lecture

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



From the previous lecture

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.

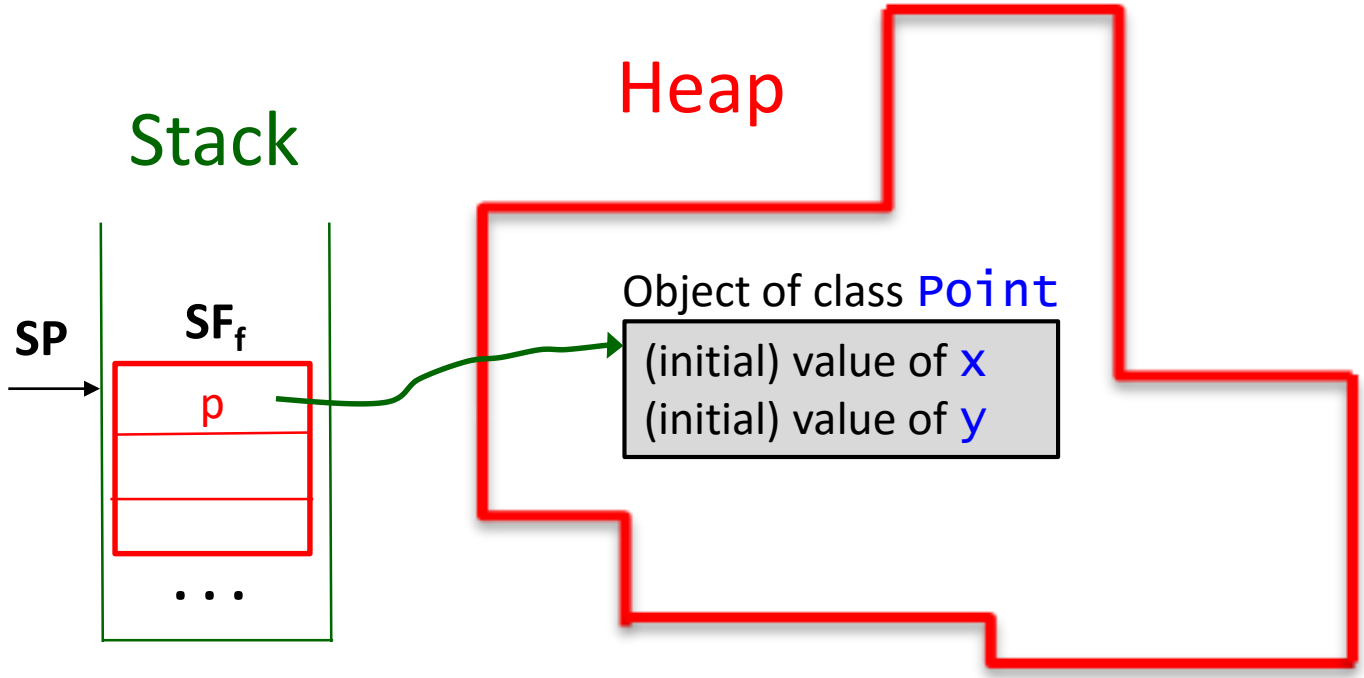
```
class Point
{
    int x;
    int y;

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

Objects and References

From the previous lecture

```
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**.

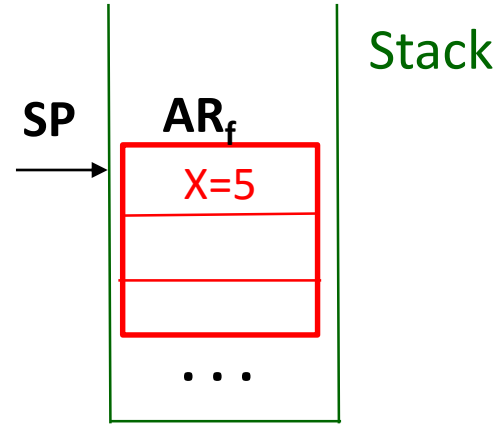


Value and Reference Types

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

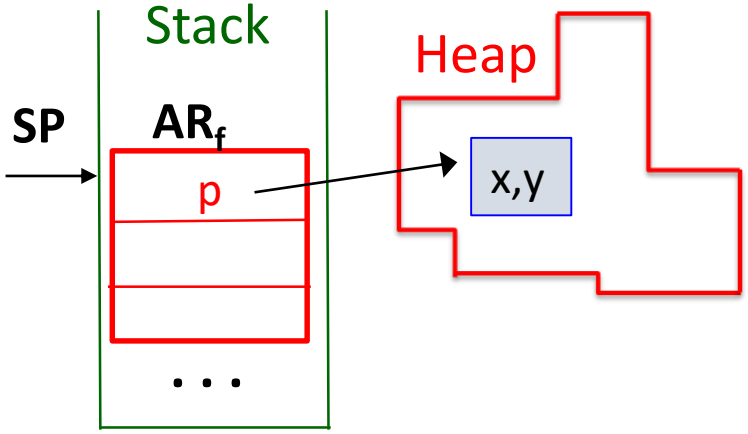
  Examples of value types: integers, floating, doubles. Values of these types are represented directly:

```
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…

Access to Class Instances

From the previous lecture

Dot notation, the common form:

```
ref_to_instance . member_name
```

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

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

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

Constructors

OR: How to initialize class instances

From the previous lecture

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);
}
```

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.

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; }
}
```

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?

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

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

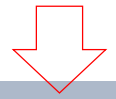
```
void printValue(int v)
{
    System.out.println(v);
}
```

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

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

Function Parameters

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

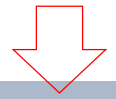


```
int sqr(int v)
{
    return v*v;
}
```

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

Function Parameters

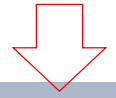
- Formal parameters (or just **parameters**)
  - Used to define functions



```
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.

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

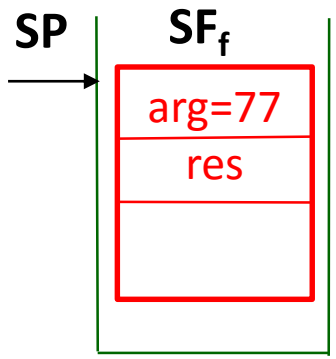
Parameter Passing in Java

- 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) ...
  }
}
```

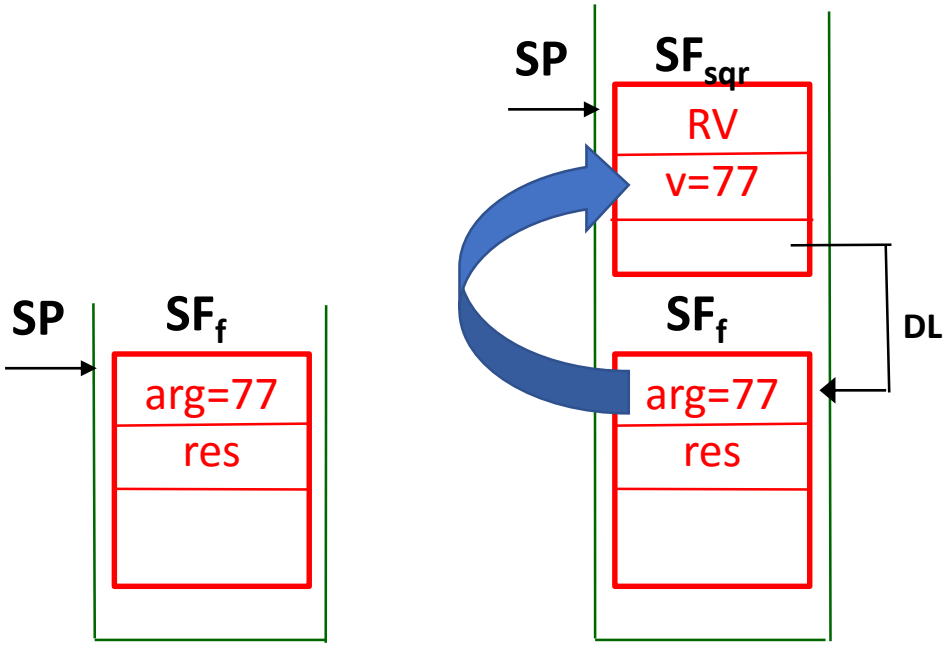


(1)

Stack

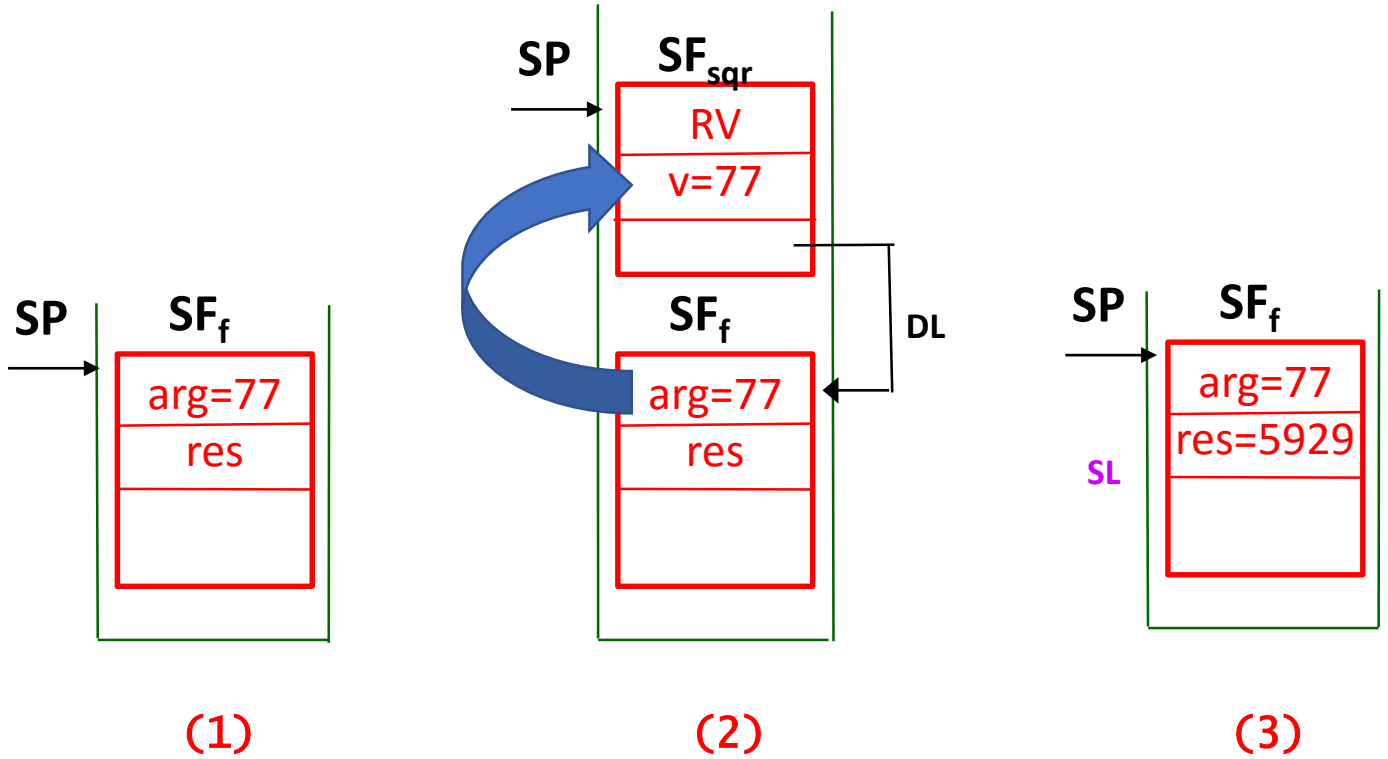
Parameter Passing in Java: Example

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

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

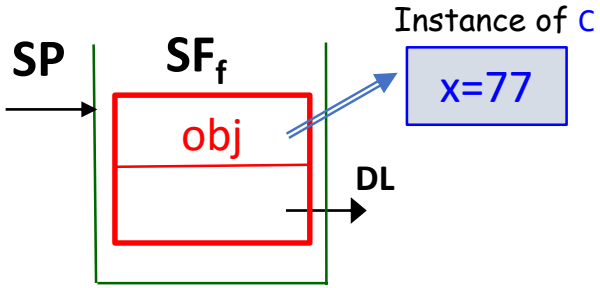


Parameter Passing in Java: Example

```
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) ...
  }
}
```

Parameter Passing in Java: Example

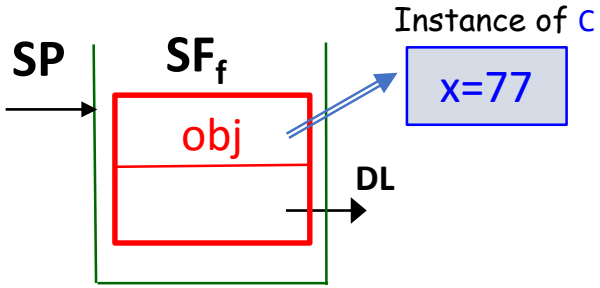
```
class C {
  public int x;
}
class Example {
  void change(C c)
  {
    c.x = 99;
  }
  void f()
  {
    C obj = new C();
(1) obj.x = 77;
(2) change(obj);
(3) ...
  }
}
```



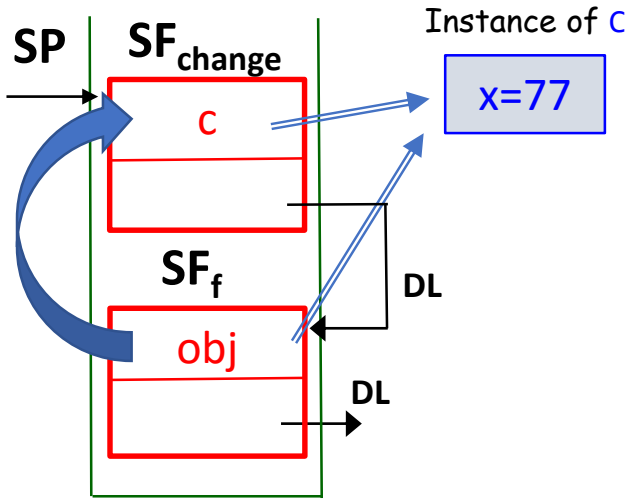
(1)

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) ...
  }
}
```



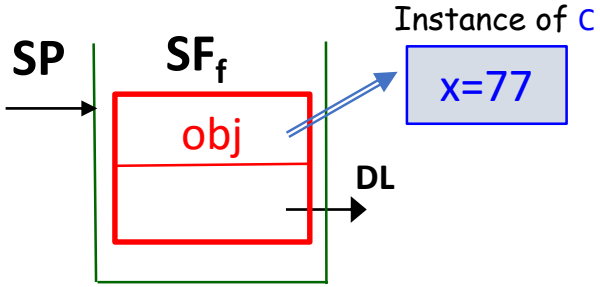
(1)



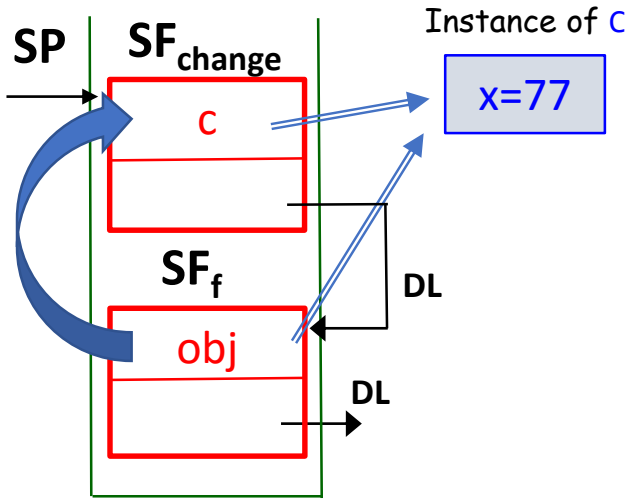
(2)

Parameter Passing in Java: Example

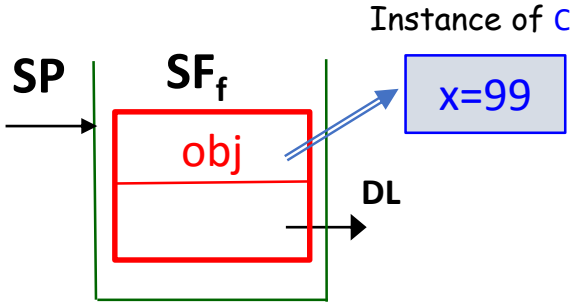
```
class C {
  public int x;
}
class Example {
  void change(C c)
  {
    c.x = 99;
  }
  void f()
  {
    C obj = new C();
(1) obj.x = 77;
(2) change(obj);
(3) ...
  }
}
```



(1)



(2)

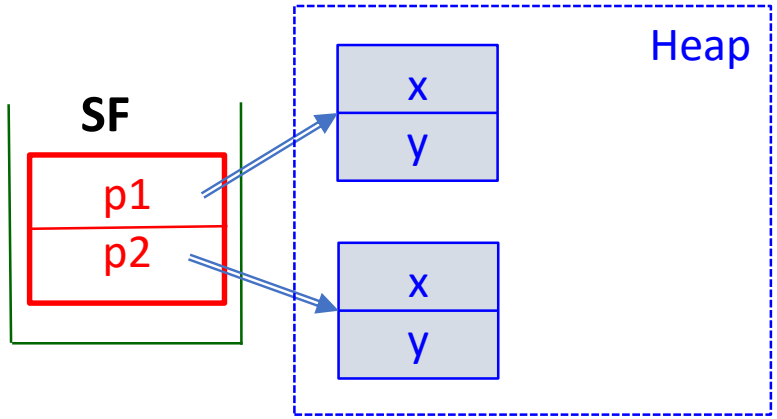


(3)

Class Attributes

- Usually, each instance has its own set of attributes

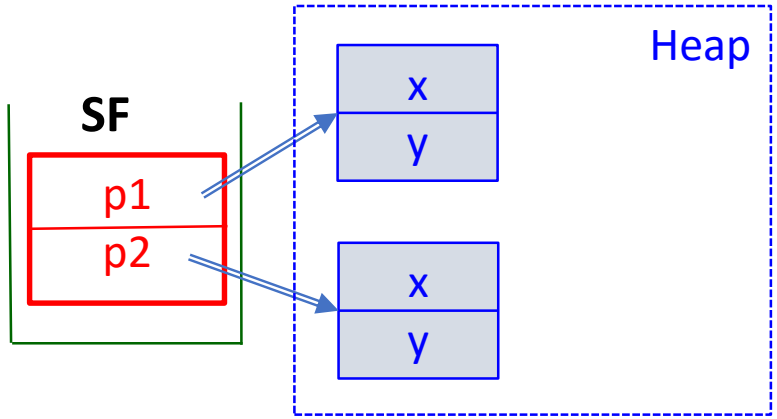
```
class Point
{
    int x, y;
}
...
Point p1 = new Point();
Point p2 = new Point();
```



Class Attributes

- Usually, each instance has its own set of attributes

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

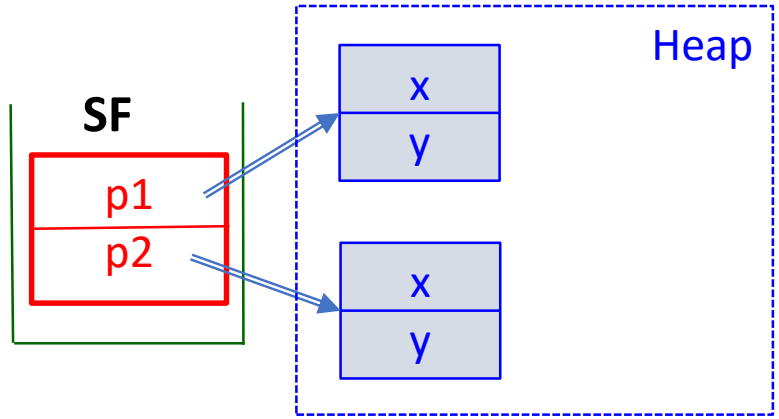


- **Class attributes**: belong to the class as a whole and are shared among all the objects of the class

Class Attributes

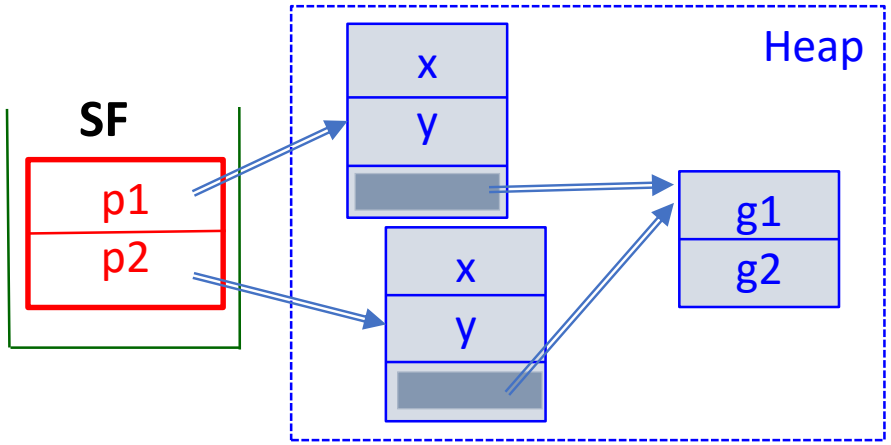
- Usually, each instance has its own set of attributes

```
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);
```

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
```

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;
   ...
}

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

Any class can contain `main`

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`.

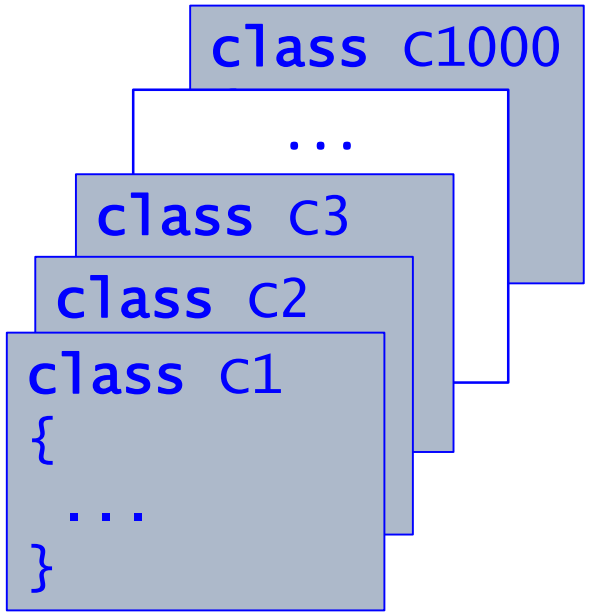
Introduction to Packages

- When developing large projects, it is essential to divide the work into cohesive units, which could be assigned to different developing teams.
  - This could lead to name conflicts, because programmers tend to use always the same names for the entities they declare.
- Moreover, in a large project it is important to organize the code in a meaningful and logical way in order to manage it more easily.

Packages address these two concerns!

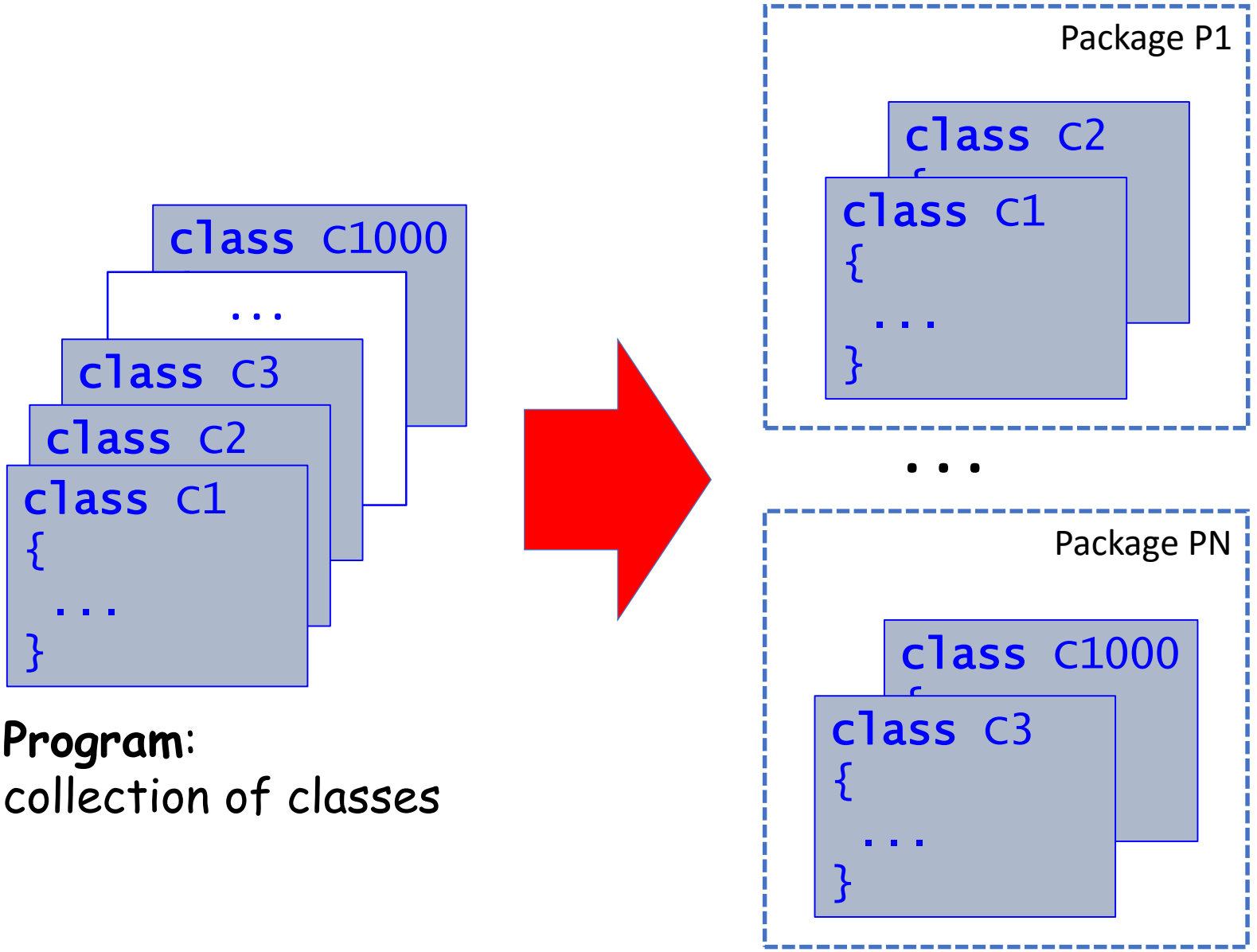
- A **package** (in the abstract sense) is a collection of related declarations providing **access protection** and **names management**.

Packages: the Idea



Program:
collection of classes

Packages: the Idea



Program:
collection of classes

The Idea of Packages in PLs

C++, C#: **namespaces**

```
namespace Part1
{
    ...
    declarations
    ...
}
```

```
namespace Part1
{
    namespace Part11
    {
        ...
        declarations
        ...
    }
}
```

The Idea of Packages in PLs

C++, C#: **namespaces**

```
namespace Part1
{
    ...
    declarations
    ...
}
```

```
namespace Part1
{
    namespace Part11
    {
        ...
        declarations
        ...
    }
}
```

Java: **packages**

```
package Part1;
...
class declarations
...
```

Packages in Java 1

- Each class or group of classes can be made a member of a package:

```
package myPackage;
class C1 { ... }
class C2 { ... }
 ...
```

This is a kind of "header" of the package called `myPackage`.

All following classes within this file are treated as members of `myPackage` package.

Full names of the classes are `myPackage.C1`, `myPackage.C2` etc. ("Fully qualified names")

Packages in Java 1

- Each class or group of classes can be made a member of a package:

```
package myPackage;
class C1 { ... }
class C2 { ... }
 ...
```

This is a kind of "header" of the package called `myPackage`.

All following classes within this file are treated as members of `myPackage` package.

Two parts of the same package

```
package myPackage;
class C10 { ... }
class C20 { ... }
 ...
```

Full names of the classes are `myPackage.C1`, `myPackage.C2` etc. ("Fully qualified names")

A package can be **made up of several files** (all residing in the same directory)

Packages in Java 2

- Packages can be **nested**:

```
package Company.Department.Lab.Math;
class C1 { ... }
class C2 { ... }
 ...
```

Here, the package `Math` is a part of package `Lab` which is a part of package `Department`, which is in turn a part of the package `Company`.

Classes `C1` & `C2` belong to the package `Math`. The fully-qualified name for `C1` is `Company.Department.Lab.Math.C1`.

Packages in Java 2

- Packages can be **nested**:

```
package Company.Department.Lab.Math;
class C1 { ... }
class C2 { ... }
 ...
```

Here, the package `Math` is a part of package `Lab` which is a part of package `Department`, which is in turn a part of the package `Company`.

Classes `C1` & `C2` belong to the package `Math`. The fully-qualified name for `C1` is `Company.Department.Lab.Math.C1`.

- Packages can manage access to their members:

```
package myPackage;
public class C1 { ... }
class C2 { ... }
 ...
```

Here, the class `C1` is visible (accessible) from outside the package `myPackage`.

Class `C2` is accessible only from classes of the package `myPackage`.

Accessing Packages 1

In general there are two ways to access a *public* entity belonging to a package:

1. The first is by using the so-called **fully qualified name.**
   - i.e. the entity name prefixed in some way by the package name.
2. The second is by using an **import directive** in the portion of code where we want to use that entity.

Accessing Packages 2

Public (and only public) classes and interfaces declared in a package are accessible from outside the package itself by using so-called import declarations:

```
import package_name . class_name ;
```

Import declarations must be put **just after the package declaration** of the current compilation unit:

```
package myPackage;
import util.math.MathVector;

public class C1 {
    MathVector v;
    ...
}
...
```

Class `MathVector` can be used inside the package `myPackage` by it short name.

Class `C1` can be used outside of the package `myPackage`: either by its fully-qualified name or by its short name (if it's imported).

Accessing Packages 3

- If we don't want to specify exactly what classes we want to import from a package, we can use the so-called **import-on-demand declaration**:

```
import package_name.* ;
```

For example, writing

```
import util.math.*;
```

we make all the classes of the package `util.math` visible in the current compilation unit.

- That's typical, for example, with the **Java libraries**, where there are lots of declarations for each package. Typical naming of Java libraries are:

```
java.lang
java.io
java.awt
```

Naming Conventions

- If a package is to be widely distributed, it is a **common convention** to prefix its name with **the reverse Internet domain name** of the producing or distributing organization, with slashes substituted by dots
  - For example, if I want to distribute a package previously named `util.math` and I work for a company having the domain name http://very.wonderful.org, then I should rename the package as
    `org.wonderful.very.util.math`
- This might potentially avoid any problem of name clashes worldwide!

Packages and File Systems

- Packages stored in a file system must be placed following a simple rule: the name of the package is to be interpreted as the (relative) path of the package in the file system.
- Dots "`.`" becomes slashes "`/`", backslashes "`\`" or whatever directory name separator your system uses
  - For example, if I want to store the package `very.util.math` on my HD under Windows, I have to put it in the directory `base_dir\very\util\math` where `base_dir` is an arbitrary directory.

Details concerning relationships between fully-qualified class names and corresponding directories and files in a file system is to be explained on labs.

Exercise 1

- Develop class representing **lines**.

  A line can be defined by **two points** representing its begin and end. Something like as follows:

```
class Line {
  Point begin;
  Point end;

  public Line(Point b, Point e)
  {
    ...
  }
  ...
}
```

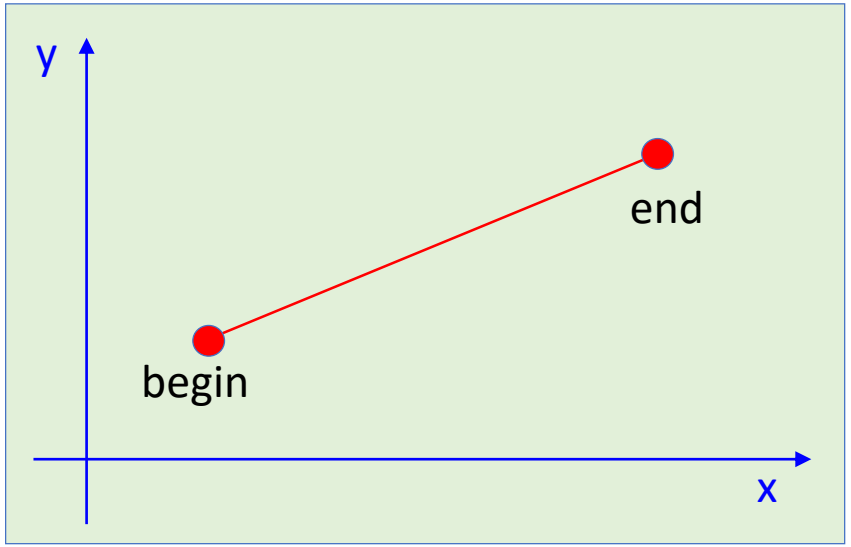
Important remarks:

- The class `Point` should be completely implemented as well.
- It's a good idea to implement the more functionality the better in the `Point` class.

- Write things necessary for manipulating lines: creation (constructor, perhaps, several ones), and the following interesting methods:

Exercise 1

length should calculate the length of the line

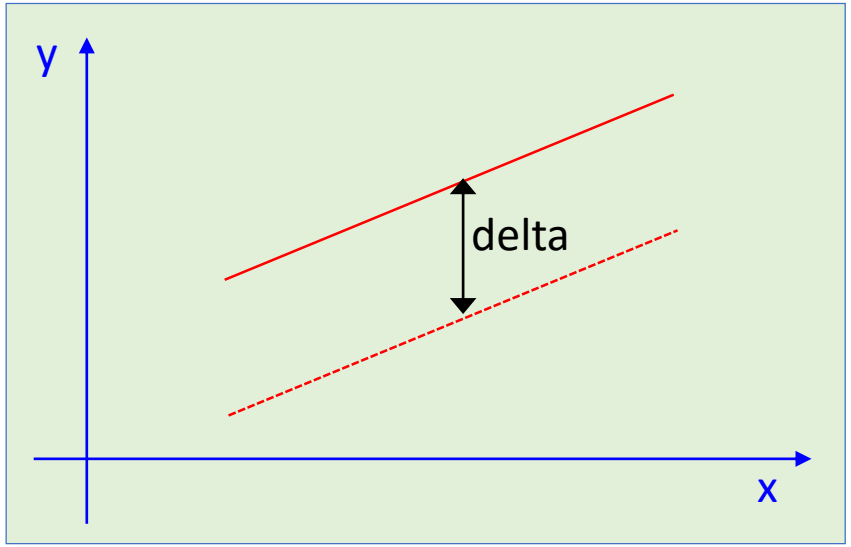



```
class Line
{
  Point begin;
  Point end;

  public int length()
  {
    ...
  }
  ...
}
```

Exercise 1

moveX
moveY

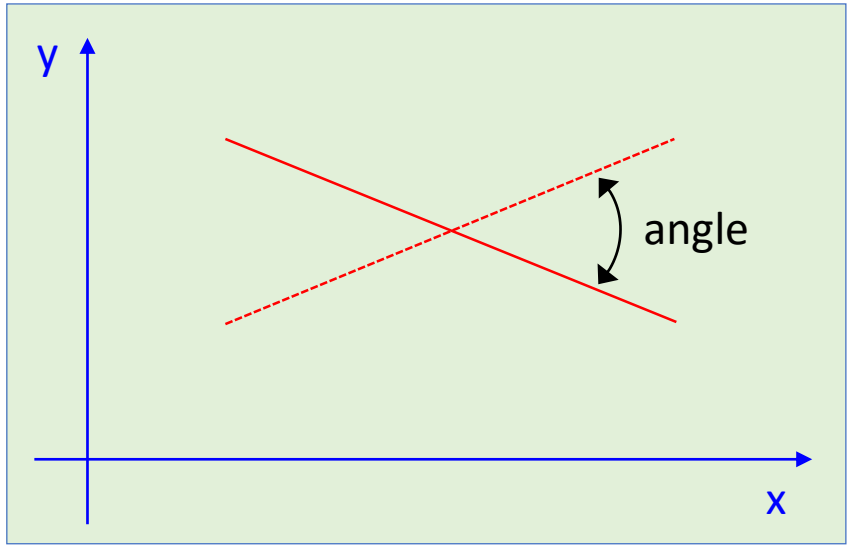
should move the line to one of directions (left, right, up and down) by the given «delta»:




```
class Line
{
  Point begin;
  Point end;

  public void moveX(int delta)
  {
    ...
  }
  public void moveY(int delta)
  {
    ...
  }
  ...
}
```

Exercise 1

`rotate` should rotate the line by its center by the given «angle»:



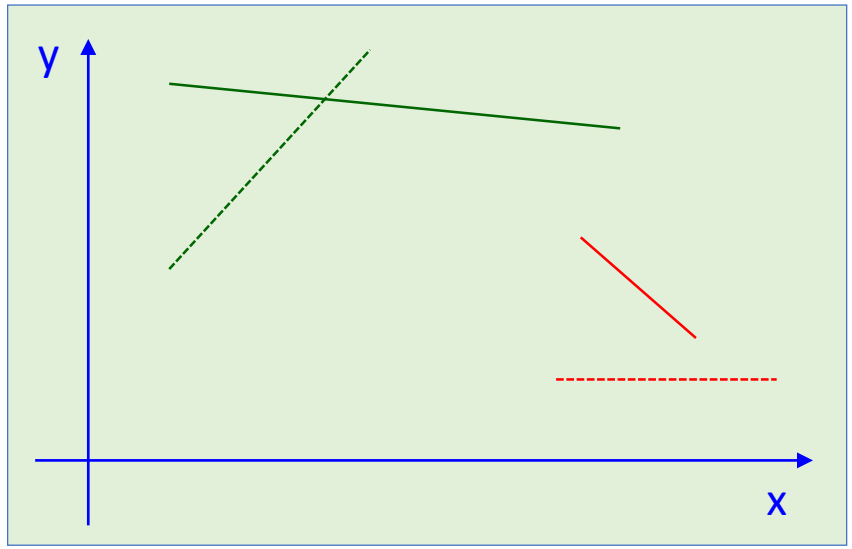

```
class Line
{
  Point begin;
  Point end;

  public void rotate(int angle)
  {
    ...
  }
  ...
}
```

It's assumed that the «angle» has the meaning of «degrees» and should be an integer within the range 0..360.

Exercise 1

`intersectsWith`

should return a boolean value indicating whether the line does intersect another line which is given via parameter:




```
class Line
{
  Point begin;
  Point end;

  public Boolean
           intersectsWith(Line other)
  {
    ...
  }
  ...
}
```

Exercise 2

Implement two design approaches for **calculating roots of a quadratic equation**.

The first approach is the conventional one: a single function.

The second approach assumes that there is a class comprising all information related to quadratic equation: its coefficients, the method calculating roots, and the roots themselves.

Exercise 2

The first approach is the conventional one:
a single function.

```
...
public void calcRoots(double a, double b, double c)
{
  if ( a == 0 ) return; // not a quadratic equation
  double d = b*b - 4*a*c;
  if ( d < 0 ) return; // A bug
  d = sqrt(d);
  double a2 = a*2;
  System.out.println((-b-d)/a2);
  System.out.println((-b+d)/a2);
}
...
```

2. The second approach involves three classes each of each abstracts one concept: coefficients, roots, and calculating actions.

The function calculating roots accepts an instance with coefficients as parameter, and returns an instance with calculated roots as the result.

```
class Quadratic
{
   double a, b, c;
   double r1, r2;

   public double root1() { return r1; }
   public double root2() { return r2; }

   public Quadratic(double a, double b, double c)
   {
      // initializing coefficients
      this.a = a; this.b = b; this.c = c;
   }
   public void calcRoots()
   {
      // calculating roots
      ...
   }
}
```

```
...
Quadratic q = new Quadratic(1,2,3);
q.calcRoots();
System.out.println(q.root1());
System.out.println(q.root2());
...
```