Introduction to Programming

Part I

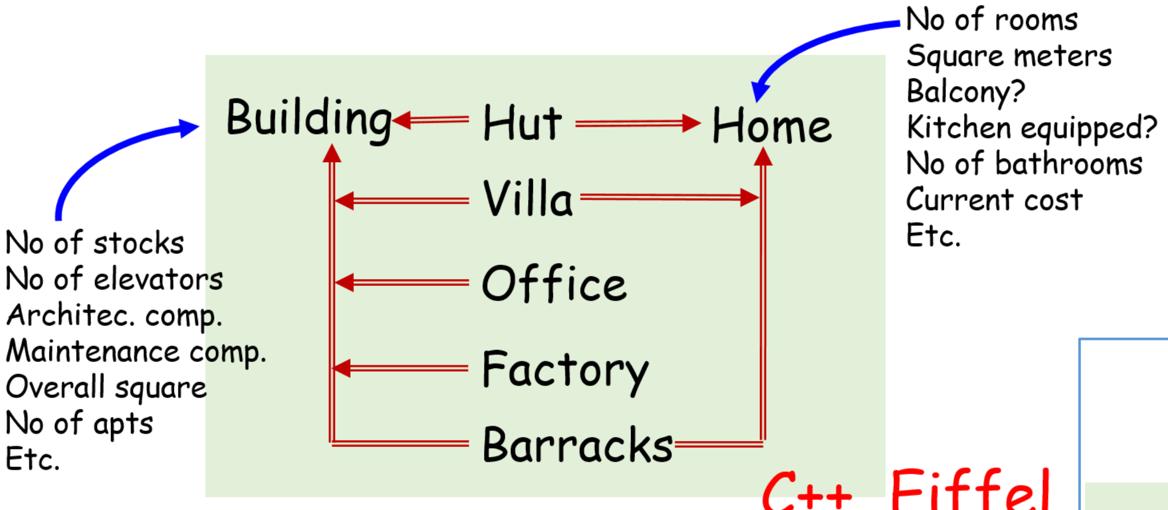
Lecture 9
Introduction to Java
Polymorphism & Type chacks

Eugene Zouev
Fall Semester 2021
Innopolis University

What We Have Learnt

- Classes and class instances
- Value types and reference types
- Encapsulation, overloading
- Inheritance: single & multiple
- Static & dynamic types
- Method overriding
- Polymorphism

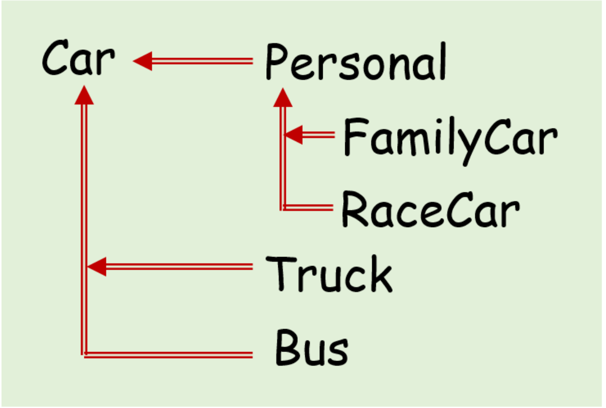
Multiple Inheritance



C++, Eiffel

"Villa" **is a** "Building" and **is** "Home" at the same time

Inheritance 3



Inheritance can be treated as **"is a"** relation:

"Personal" **is a** "Car"
"FamilyCar" **is** "Personal"
"FamilyCar" **is a** "Car"

Another kind of relation is **delegation**: **"has a"** relation:

"Car" **has an** "engine". Therefore, "Personal" and "FamilyCar" also **have an** "Engine" - as all other kinds of "Cars".

Static & Dynamic Types 2

Static type of `figure` is `Shape`: it is specified statically, in the program text.

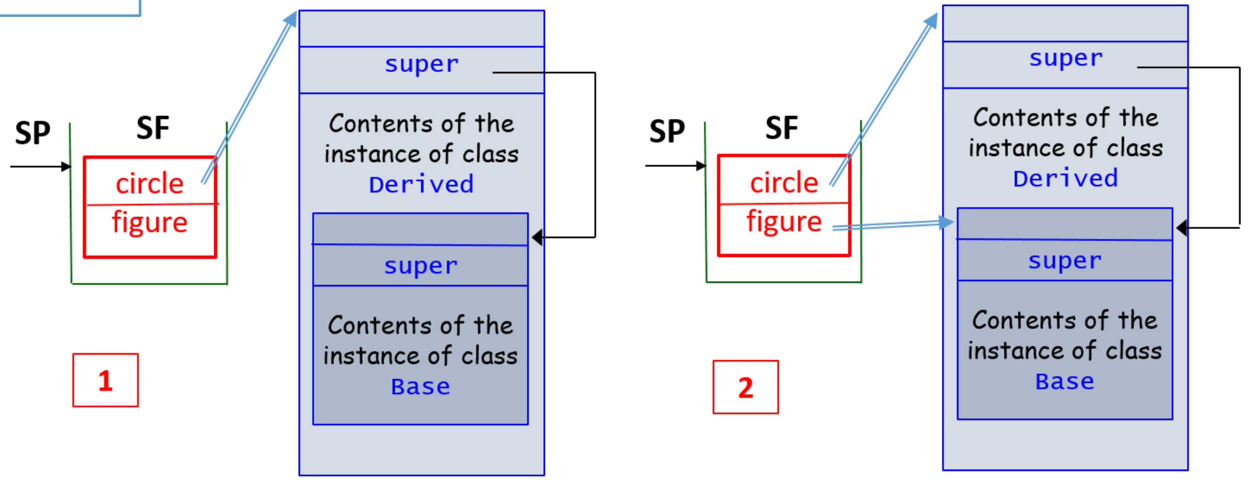
```
Circle circle = new Circle();
...
Shape figure = circle;
```

This is the conversion:
from derived type to base type

After this assignment `figure` refers to an instance of class `Circle`. It's said, that the **dynamic type** of `figure` **now** is `Circle`.

Static & Dynamic Types 3

```
(1)    Circle circle = new Circle();
       ...
(2)    Shape figure = circle;
```



The main rule of polymorphism

The interpretation of the call of a virtual method **depends on the type of the object for which it is called (the dynamic type)**,

whereas

the interpretation of a call of a non-virtual method function depends only on the type of the reference denoting that object (the **static type**).

Polymorphism 7

Late binding

```
class Base
{
  public int f(int p) { return x*x; }
}

class Derived extends Base
{
  public int f(int p) { return x*x*x; }
}
```

These two methods have the same signature

This method overrides the method with the same signature from the base class

```
class SomeOtherClass
{
  public void someOtherMethod()
  {
    int result;
    Base m = new Base(); result = m.f(3);
    m = new Derived();   result = m.f(3);
  }
}
```

Here, the dynamic type of m is Base. The method f from Base gets called

The static type of m is (always) Base

Here, the dynamic type of m is Derived. The method f from Derived gets called!

32/35

What's For Today

- Upcasting, downcasting & type checks
- Abstract classes & methods
- Packages

Downcasting & Type Checks 1

Upcasting:
Each Lion is an Animal
This relation is **always true** => conversion form Lion to Animal is always correct and safe.

Downcasting:
If **this** particular Animal **is actually** a Lion (if we know this for sure ☺) then the cast to the derived class is correct and safe.

```
class Animal { ... }

class Lion extends Animal { ... }
class Frog extends Animal { ... }
 ...
Animal a = new Frog();
 ...
a = new Lion();
... (Lion)a ...
```

Downcasting & Type Checks 1

Upcasting:
Each Lion is an Animal
This relation is **always true** => conversion form Lion to Animal is always correct and safe.

Downcasting:
If **this** particular Animal **is actually** a Lion (if we know this for sure ☺) then the cast to the derived class is correct and safe.

a can refer to an object of class Animal OR to an object of its derived class

```
class Animal { ... }

class Lion extends Animal { ... }
class Frog extends Animal { ... }
 ...
Animal a = new Frog();
 ...
a = new Lion();
... (Lion)a ...
```

Downcasting & Type Checks 1

Upcasting:
Each Lion is an Animal
This relation is **always true** => conversion form Lion to Animal is always correct and safe.

Downcasting:
If **this** particular Animal **is actually** a Lion (if we know this for sure ☺) then the cast to the derived class is correct and safe.

a can refer to an object of class Animal OR to an object of its derived class

Here we know for sure that a refers to the object of class Lion (i.e., the dynamic type of a is Lion) => the cast is safe

```
class Animal { ... }

class Lion extends Animal { ... }
class Frog extends Animal { ... }
 ...
Animal a = new Frog();
 ...
a = new Lion();
... (Lion)a ...
```

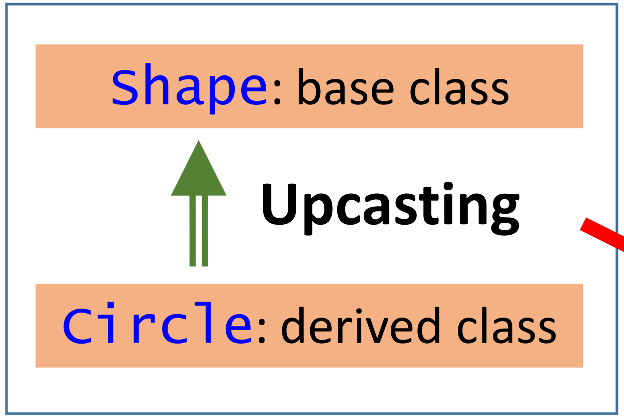
Downcasting & Type Checks 2

```java
class Shape { ... }

class Circle extends Shape { ... }
```

```java
Circle circle = new Circle();
...
Shape figure = circle;
...
Circle c2 = (Circle)figure;
```

Downcasting & Type Checks 2

```
class Shape { ... }

class Circle extends Shape { ... }
```




```
Circle circle = new Circle();
...
Shape figure = circle;
...
Circle c2 = (Circle)figure;
```

Basic OOP rule:

- **Object of the derived type <u>can be converted</u> to an object of the base type**

The rule is based on the relation "is a":

Circle **is a** Shape hence Circle can be treated as Shape.

Downcasting & Type Checks 2

```
class Shape { ... }

class Circle extends Shape { ... }
```

Shape: base class

Upcasting

Circle: derived class

```
Circle circle = new Circle();
...
Shape figure = circle;
...
Circle c2 = (Circle)figure;
```

Shape: base class

Downcasting

Circle: derived class

Basic OOP rule:

- **Object of the derived type <u>can be converted</u> to an object of the base type**

The rule is based on the relation “is a”:
Circle **is a** Shape hence Circle can be treated as Shape.

<u>Upcasting</u>: always valid

<u>Downcasting</u>: valid only if the instance is actually of the target type

Downcasting & Type Checks 3

Type check operator: `instanceof`

```
obj instanceof Class
```

RTTI:
run-time type identification

Returns **`true`** if dynamic type of `obj` is `Class` OR **any of its derived classes**, and **`false`** otherwise

Downcasting & Type Checks 3

Type check operator: `instanceof`

```
obj instanceof Class
```

RTTI:
run-time type identification

Returns `true` if dynamic type of `obj` is `Class` OR **any of its derived classes**, and `false` otherwise

```
class Animal { ... }
class Lion extends Animal { ... }
...
Animal a = new Lion();
boolean r1 = a instanceof Animal;      // true
boolean r2 = a instanceof Lion;        // true
boolean r3 = a instanceof Car;         // false
```

C++: `typeid` operator (*not exactly the same*)
C#: `is` operator (!!!)

Downcasting & Type Checks 4

Static type of a is `Animal`.
a can refer to an object of types `Animal`, `Lion`, or `Frog`.

```
class Animal { public int f1; }
class Lion extends Animal { public int f2;}
class Frog extends Animal { public int f3;}

Animal a = new Lion();
...
a = new Frog();
...
if (a instanceof Lion)
   // Downcasting is safe here
   ...((Lion)a).f1...
else if (a instanceof Frog)
   ...((Frog)a).f3...
```

Downcasting & Type Checks 4

```
class Animal { public int f1; }
class Lion extends Animal { public int f2;}
class Frog extends Animal { public int f3;}

Animal a = new Lion();
...
a = new Frog();
...
if (a instanceof Lion)
    // Downcasting is safe here
    ...((Lion)a).f1...
else if (a instanceof Frog)
    ...((Frog)a).f3...
```

Static type of a is Animal. a can refer to an object of types Animal, Lion, or Frog.

Here, a is treated as Lion. Therefore, features from Lion (and, of course, Animal) are accessible via a.

Static type of a is (still) Animal. Actually, a refers to the object of type Frog. The dynamic type of a is Frog.

Abstract Classes & Methods 1

An informal introduction from Prof Giancarlo Succi:

Sometimes, a class that you define represents an abstract concept and, as such, should not be instantiated.

Take, for example, **food** in the real world. Have you ever seen an instance of food? No. What you see instead are instances of carrot, apple, and (our favorite) chocolate.

Food represents the abstract concept of things that we all can eat. It doesn't make sense for an instance of food to exist.

Abstract Classes & Methods 1

An informal introduction from Prof Giancarlo Succi:

Sometimes, a class that you define represents an abstract concept and, as such, should not be instantiated.

Take, for example, **food** in the real world. Have you ever seen an instance of food? No. What you see instead are instances of carrot, apple, and (our favorite) chocolate.

Food represents the abstract concept of things that we all can eat. It doesn't make sense for an instance of food to exist.

(Zouev's addition ☺)
However we know for sure that each kind of food has some **common features**: attributes & behavior. For example, "caloricity", ingredients, the way of cooking etc. We know nothing about "caloricity of food" (it's just an *abstract* feature), but know caloricity of apple…

Abstract Classes & Methods 2

So the **conclusion** is:
If you are going to represent an abstract notion in your program, think about making the corresponding class **abstract**.

```
abstract class Vehicle
{
   // Features that are common
   // to all possible vehicles
   Color color;
   int numWheels;
   ...
   abstract void startEngine();
   ...
}
```

Abstract Classes & Methods 2

So the **conclusion** is:
If you are going to represent an abstract notion in your program, think about making the corresponding class **abstract**.

```
abstract class Vehicle
{
  // Features that are common
  // to all possible vehicles
  Color color;
  int numWheels;
  ...
  abstract void startEngine();
  ...
}
```

We **cannot create instances of abstract classes**: what does it mean “an instance of a vehicle”?

In the abstract class we can define behavior of each categories of vehicles – without any detalization (no body)
These are **abstract methods**.

Abstract Classes & Methods 2

So the **conclusion** is:
If you are going to represent an abstract notion in your program, think about making the corresponding class **abstract**.

```
abstract class Vehicle
{
  // Features that are common
  // to all possible vehicles
  Color color;
  int numWheels;
  ...
  abstract void startEngine();
  ...
}
```

```
abstract class Car extends Vehicle
{ ... }
```

We **cannot create instances of abstract classes**: what does it mean "an instance of a vehicle"?

In the abstract class we can define behavior of each categories of vehicles – without any detalization (no body)
These are **abstract methods**.

Classes representing "real" vehicles are declared as derived classes. They can be "usual" classes OR in turn abstract ones!

Abstract Classes & Methods 3

Some remarks & details

- One could correctly argue that deriving from **class Vehicle** is only a way to logically group objects of the derived classes.
- No "Vehicle" objects exist in real life: we have cars, planes, trains, bikes, etc., but no "generic" vehicles.
- Java, C#, C++: abstract classes;
  Eiffel: deferred classes.
- Java, C#: abstract methods;
  C++: ***pure virtual*** methods.
- A class that is declared abstract does not have to have abstract methods in it.
- A class containing an abstract method must be declared abstract.

Abstract Classes & Methods 4

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This “preliminary” declaration is only to tell the developer of derived classes that *the implementation of the method* ***is required*** *in all direct subclasses that want to become instantiable.*

Abstract Classes & Methods 4

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This "preliminary" declaration is only to tell the developer of derived classes that *the implementation of the method **is required** in all direct subclasses that want to become instantiable.*

```
class Motobike extends Vehicle
{
    ...
    void startEngine()
    {
      // real algorithm
    }
}
```

If the derived class provides implementations **for all** abstract methods from its superclass then this derived class **is not considered abstract**. – It's a "real" class…

Abstract Classes & Methods 4

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This "preliminary" declaration is only to tell the developer of derived classes that *the implementation of the method **is required** in all direct subclasses that want to become instantiable.*

```
class Motobike extends Vehicle
{
    ...
    void startEngine()
    {
      // real algorithm
    }
}
```

If the derived class provides implementations **for all** abstract methods from its superclass then this derived class **is not considered abstract**. – It's a "real" class…

```
Motobike my = new Motobike();
```

…and we can create instances of this class.

Abstract Classes & Methods 5

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This "preliminary" declaration is only to tell the developer of derived classes that *the implementation of the method **is required** in all direct subclasses that want to become instantiable.*

Abstract Classes & Methods 5

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This "preliminary" declaration is only to tell the developer of derived classes that *the implementation of the method **is required** in all direct subclasses that want to become instantiable.*

```
class FlyingVehicle extends Vehicle
{
    ...
// void startEngine()
// {
//    // real algorithm
// }
}
```

If the derived class **doesn't provide** implementations for some abstract methods from its superclass then this derived class **is still considered abstract**…

Abstract Classes & Methods 5

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This "preliminary" declaration is only to tell the developer of derived classes that *the implementation of the method **is required** in all direct subclasses that want to become instantiable.*

```
class FlyingVehicle extends Vehicle
{
    ...
// void startEngine()
// {
//    // real algorithm
// }
}
```

If the derived class **doesn't provide** implementations for some abstract methods from its superclass then this derived class **is still considered abstract**...

```
FlyingVehicle my =
     new FlyingVehicle(); // ERROR
```

...and we **cannot** create instances of this class.

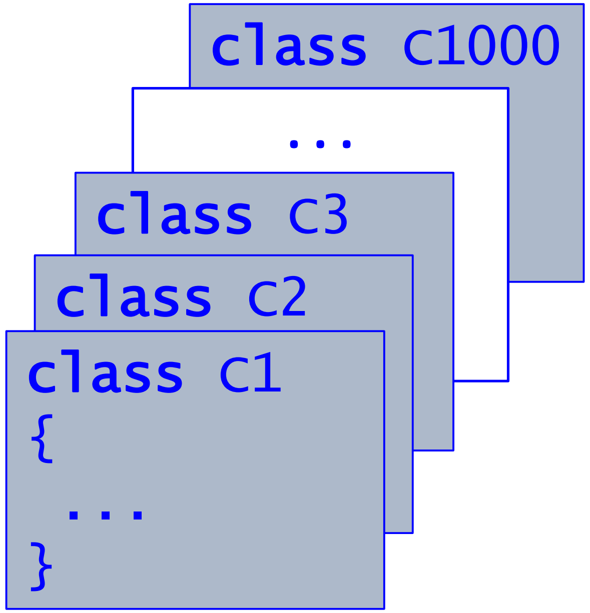
Introduction to Packages

- 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!

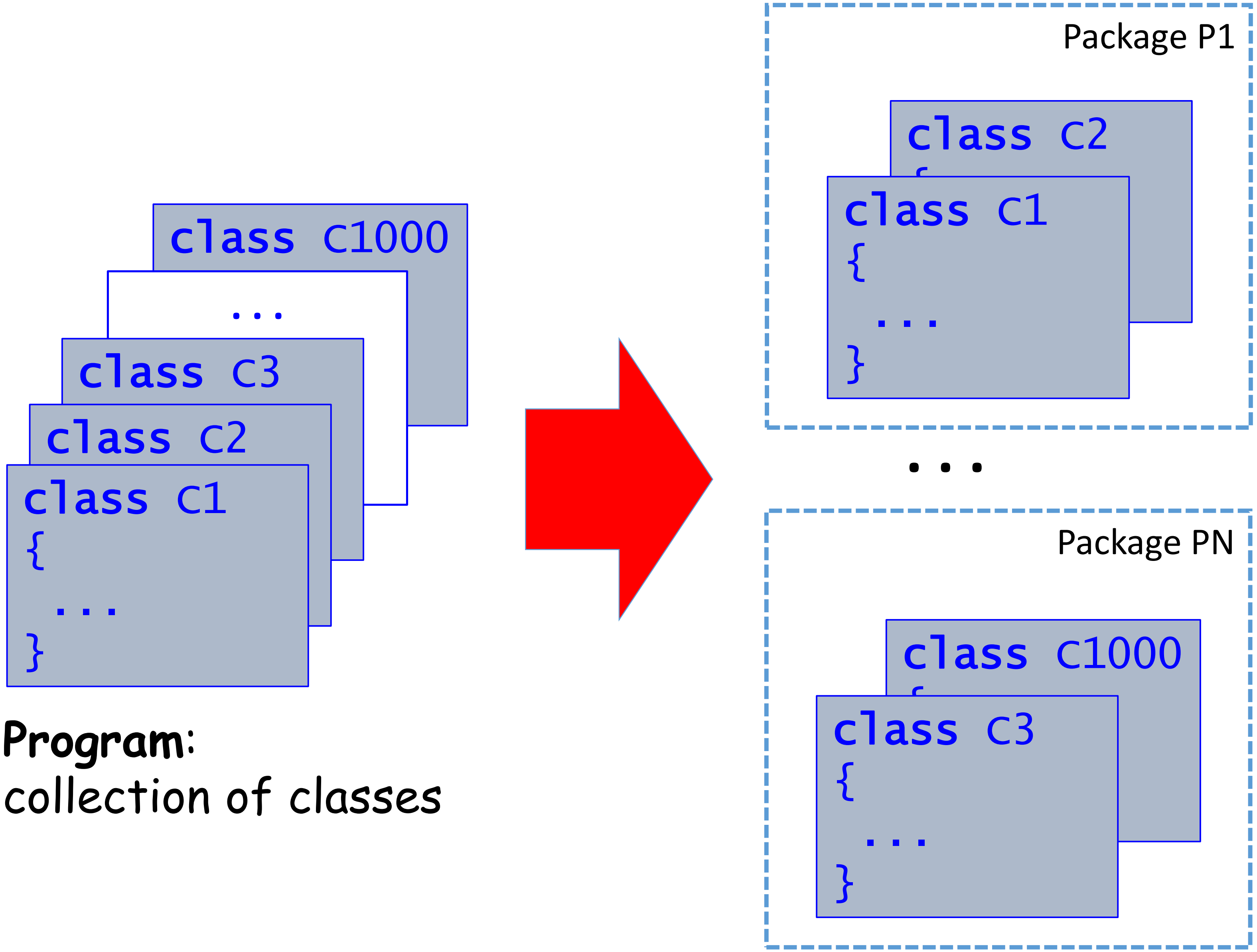
- A **package** (in the abstract sense) is a collection of related declarations providing **access protection** and **names management**.

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.a
```

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.

Accessibility Rules 1

Version 1

```
class Base
{
   public int m1;
}
```

- Here, `m1` is accessible from any other class.

Accessibility Rules 1

```java
class Base                Version 1
{
    public int m1;
}
```

```java
class Base                Version 2
{
    int m1;
}
```

- Here, `m1` is accessible from any other class.
- Suppose we remove **`public`** specifier. Then, `m1` becomes accessible only within `Base`'s package, but still from any other class.

Accessibility Rules 1

```
class Base                     Version 1
{
    public int m1;
}
```

- Here, `m1` is accessible from any other class.

```
class Base                     Version 2
{
    int m1;
}
```

- Suppose we remove **`public`** specifier. Then, `m1` becomes accessible only within `Base`'s package, but still from any other class.

```
class Base                     Version 3
{
    private int m1;
}
```

- Next option: let's make `m1` **`private`**. Then, `m1` becomes inaccessible everywhere except its own class - hence, inaccessible within the derived class.

Accessibility Rules 1

Version 1

```
class Base
{
    public int m1;
}
```

Version 2

```
class Base
{
    int m1;
}
```

Version 3

```
class Base
{
    private int m1;
}
```

Version 4

```
class Base
{
    protected int m1;
}
```

- Here, `m1` is accessible from any other class.
- Suppose we remove **`public`** specifier. Then, `m1` becomes accessible only within `Base`'s package, but still from any other class.
- Next option: let's make `m1` **`private`**. Then, `m1` becomes inaccessible everywhere except its own class - hence, inaccessible within the derived class.
- To provide member's accessibility only within derived classes, the special specifier is introduced: **`protected`**.

Accessibility Rules 2

- `private` members are accessible only within the class.
- `protected` members are accessible in the class and from all its derived classes, **and** from any class within the same package (i.e., where its class is declared).
- `public` members are accessible from any other class.
- Members without a specifier are **available from classes within the same package**.
- The rules affect all kinds of class members including both instance and static methods/attributes.

- `public` **classes** are accessible from any other class.
- **Classes without** `public` specifier are accessible only within the package they belong to.