Class hierarchies

Classes: sub- and superclasses
Class hierarchies: inheritance, polimorphism
Overriding inherited methods
Hiding inherited attributes
Abstract and final classes
Access control to classes and their components
Packages

Basic definition of subclasses

```
public class Person {
    public String name;
    public int age;
    public String toString() {
        return "name: " + name + "\nage: " + age;
    }
}

public class Employee extends Person {
    public long grossSalary;
    public Manager boss;
    ...
}

public class Manager extends Employee {
    public long incentive;
    public ArrayList<Employee> team;
    public void setIncentive(long c) { incentive = c; }
}
```

*Employee is a **subclass** of Person;*

*Person is the **superclass** of Employee*

Root class of the hierarchy

Which class is the root of the hierarchy?
The predefined class **`Object`**

The previous definition of the **`Person`** class (without explicit superclass) is equivalent to the following one:

```
public class Person extends Object {
    public String name;
    public int age;
    public String toString() {
        return "name: " + name + "\nage: " + age;
    }
}
```

What does Object contain?

- protected Object clone() throws CloneNotSupportedException

  Creates and returns a copy of this object

- public boolean equals(Object obj)

  Indicates whether some other object is "equal to" this one

- protected void finalize() throws Throwable

  Called by the garbage collector on an object when garbage collection determines that there are no more references to the object

- public final Class getClass()

  Returns the runtime class of an object

- public int hashCode()

  Returns a hash code value for the object

- public String toString()

  Returns a string representation of the object

- Methods to synchronize threads (notify, …)

We will analyse this later

Type hierarchy: “is_a” relation, castings

```
Person p1, p2 = new Person();
Employee e, emp = new Employee();
Manager d, dir = new Manager();
```

- Compatibility (***generalization***)

```
p1 = emp; // Employee → Person
p2 = dir; // Manager  → Person
e  = dir; // Manager  → Employee
```

A Manager object can play the roles of Employee and Person

- Explicit conversion (***specialization***, responsibility of the programmer)

```
e = p2;           // Compilation error
d = (Manager) p2; // Person → Manager
d = (Manager) p1; // Execution error
                    // p1 is not a Manager
d = (Dog) p1;    // Compilation error (Dog is neither sub-
                  // or super-class of Person)
```

*A Person object can play the role of Manager only **if it really is a** Manager*

Type hierarchy: argument compatibility and conversion

```
class Corporation {
    void f(Employee p) { ... }
    void g(Manager p) { ... }
}
```

```
Manager dir    = new Manager();
Employee e     = dir, emp = new Employee();
Corporation c = new Corporation();
```

- Implicit conversion (*towards more general types*)

```
c.f(dir); // Manager → Employee
```

- Explicit castings

```
c.g(e);                // Compilation error
c.g((Manager) e);      // Employee → Manager
c.g((Manager) emp);    // Execution error: emp is not a
                       // Manager
```

Inheritance of methods and attributes

Methods and attributes (defined or inherited) in the superclass are inherited by the subclasses

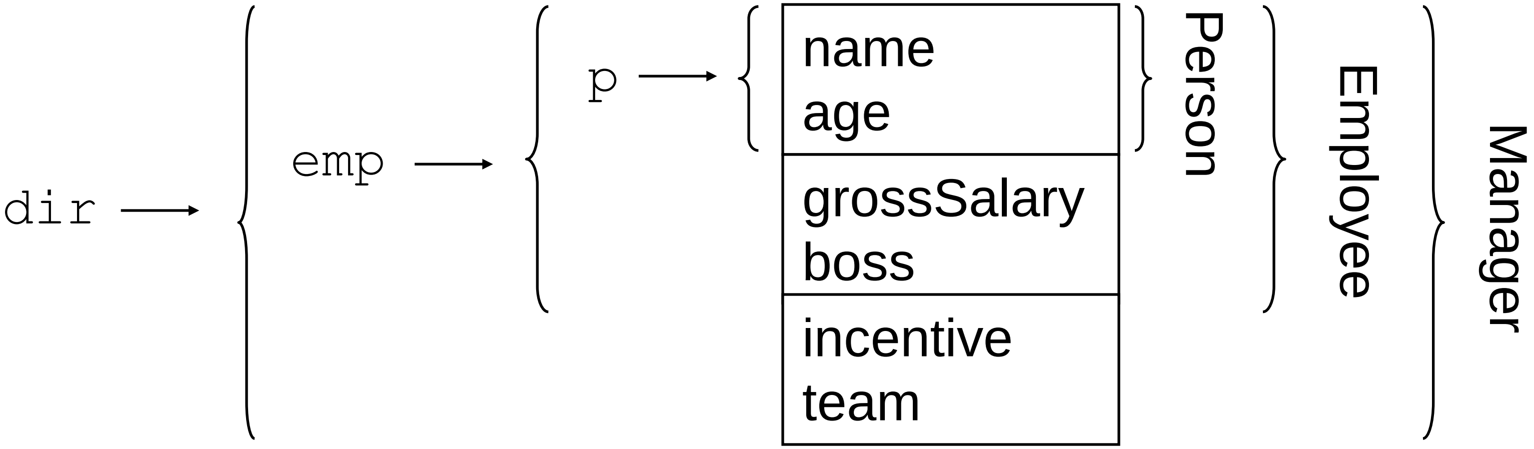
```
Employee emp = new Employee();
Manager  dir = new Manager();

emp.name = "Pedro";       // attribute of emp inherited from Person
emp.age = 28;
emp.grossSalary = 2000;   // attribute of emp defined in Employee
emp.boss = dir;
System.out.println(emp); // Implicit call to emp.toString()
                          // method defined in Person
dir.name = "Maria";
dir.age = 45;
dir.grossSalary = 5000;
dir.boss = null;
dir.incentive = 1500;
System.out.println(dir.toString());
```

Inheritance and type hierarchy

```
Manager  dir = new Manager();
Employee emp = dir;
Person   p   = dir;
```

Declared vs
Run-time type

```
p.grossSalary = 1000; // Error: grossSalary not
                      // definided for Person
emp.setIncentive(0);  // Error: setIncentive not defined
                      // for Employee
```



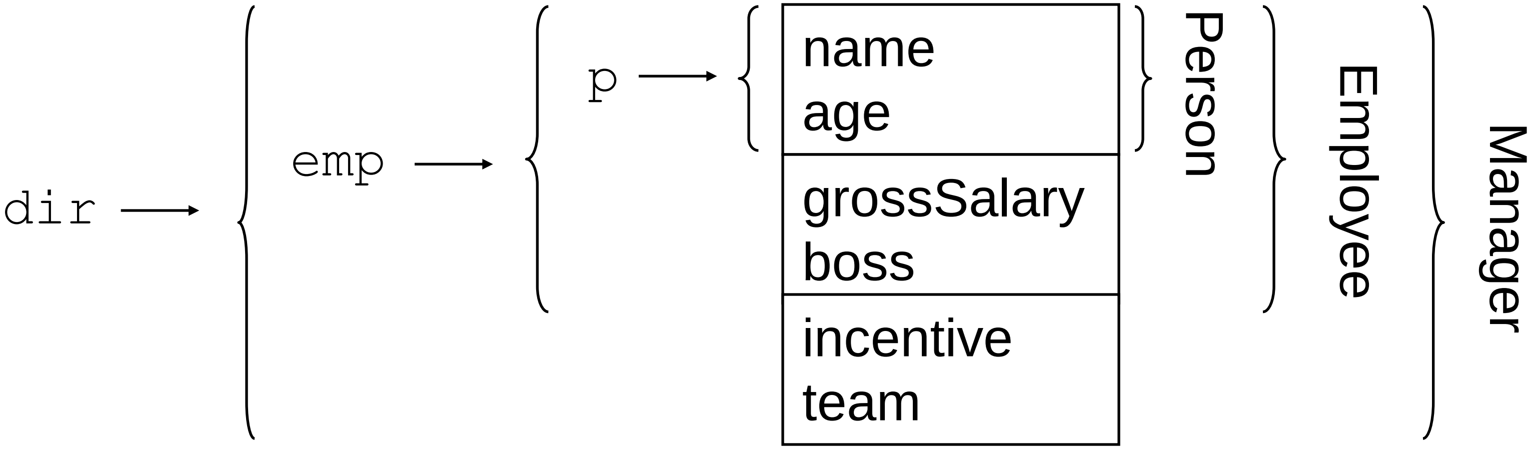
Inheritance and type hierarchy

```
Manager dir = new Manager();
Employee emp = dir;
Person p = dir;
```

Declared vs
Run-time type

```
((Employee)p).grossSalary = 1000; // OK
((Manager)p).grossSalary = 1000; // OK
((Manager)emp).setIncentive(0); // OK
```

Castings should be avoided whenever possible



Inheritance also occurs with `private`

```
public class Person {
    private String name;
    private int    age;
    public  String toString() {
        return "name: " + name + "\nage: " + age;
    }
}

public class Employee extends Person {
    private long grossSalary;
    private Manager boss;
    ... // methods defined in Employee… needed now ?!
}

public class Manager extends Employee {
    private long incentive;
    private ArrayList<Employee> team;
    public void setIncentive(long c) { incentive = c; }
}
```

Protected: to access the attributes of the superclass from the subclass

Method overriding

- A subclass may override methods inherited from the superclass
- The definition in the subclass takes precedence over the one in the superclass
- The superclass method definition is accesible using variable **`super`**
- Method overriding = specialize in the subclass an inherited method
  - The overriding method definition should have same name, and same argument types
  - Covariant return: the overriding method can return the same type, or a subtype
  - If the argument types do not match, it is a method overload, where both methods coexist (e.g.: “A” **+** “2” vs. 3 **+** 2)
  - We cannot increase privacity when overriding (but it is possible to go from protected to public, or from package to protected or public)
- This technique facilitates extensibility and reuse, taking advantage of dynamic binding
- The proliferation of method identifiers is reduced

Method overriding

```
public class Employee extends Person {
    long grossSalary;
    Manager boss;
    public String toString() {
        return "name: " + name + "\nage: " + age +
               "\nSalary: " + grossSalary + "\nboss: " +
               ((boss == null)? name : boss.name);
    }
}

// main block
Employee emp = new Employee();
Person p = emp;
emp.toString(); // toString from Employee
p.toString();   // toString from Employee (dynamic binding)
```

Remark: **`toString()`** from Person overrides the **`toString()`** method from **`Object`**

Method overriding

```java
public class Employee extends Person {
    long grossSalary;
    Manager boss;
    @Override
    public String toString() {
        return "name: " + name + "\nage: " + age +
               "\nSalary: " + grossSalary + "\nboss: " +
               ((boss == null)? name : boss.name);
    }
}
```

- Overriding methods can be explicitly annotated with **@Override**
- This helps to detect errors in the declaration of an overriding method

Can we improve this method?

Method overriding

```
public class Employee extends Person {
    long grossSalary;
    Manager boss;
    long netSalary() {
        long result;
        result =… // calculation of the net salaty
        return result;
    }
}
```

Sometimes we can reuse the code of the superclass method

```
public class Manager extends Employee {
    long incentive;
    …
    long netSalary() {
        return super.netSalary() + incentive;
    }
}
```

Method overriding. Covariant return

```
class C {
  public void print() { System.out.println("C"); }
}
class D extends C{
 @Override public void print() { System.out.println("D"); }
}

class A {
  public C f() { return new C();}
}
class B extends A {
  @Override public D f() { return new D();}
}

public class Test1 {
  public static void main(String[] args) {
    new B().f().print();    // writes D
    new A().f().print();    // writes C
    A a = new B();
    a.f().print();          // What does it write?
  }
}
```

Co-variant return

Co-variant return. Why does it work?

```
public class Test1 {
  public static void main(String[] args) {
    A aes[] = {new B(), new A()};
    // We can safely assign the return of f() to a variable
    // of the typed returned by f() in the superclass
    for (A a : aes) {
      C c = a.f();  // This assignment is safe
      c.print();
    } // prints D C
  }
}
```

What problem would we have if the opposite kind of return (contra-variant) were allowed?

Exercise (from last year’s partial exam)

We need to build an application for the management of a store of second hand items.

The warehouse manages products, which have a name, a price in euros, and a discount that can be either a configurable percentage of the total price, or a fixed amount of euros. The discounts have a text that describes the promotion. Once a product has been created, its price and discount cannot be modified.

Using principles of object orientation, encode in Java the classes necessary for the following program to output below.

```java
package products;

public class Warehouse {
  public static void main(String[] args) {
   // p1 is a Product with price 150.0€, and 15.0% discount due to promotion "no VAT"
   Product p1 = new Product("Floor lamp", 150.0,
                                   new PercentageDiscount("No VAT", 15.0));
   // p2 is a Product with price 90.0€, and 10.0€ discount by “clearance”
   Product p2 = new Product("Cutlery 50 items", 90.0,
                                   new FixedDiscount("Clearance", 10.0));
   System.out.println("Products in warehouse:\n "+p1+"\n "+p2);
   System.out.println("Higher price: "+Product.higherPrice());
  }
}
```

Output:

```
Products in warehouse:
 Floor lamp price: 127.5 with discount: No VAT
 Cutlery 50 items price: 80.0 with discount: Clearance
Higher price: 127.5
```

Wrong override attempt

```
public class Point {
  private int x = 0, y = 0, color;


  int getX() { return x; }
  int getY() { return y; }
}

class RealPoint extends Point {
  double dx = 0.0, dy = 0.0;



  double getX() { return dx; }
  double getY() { return dy; }
}
```

Compilation error:

Co-variant return does not apply to primitive types (and in any cse here we do not have a co-variant return)

Design error?

Correction attempt

```
public class Point {
  private int x = 0, y = 0, color;


  int getX() { return x; }
  int getY() { return y; }
}

class RealPoint extends Point {
  double dx = 0.0, dy = 0.0;




  int getX() { return (int)Math.floor(dx); }
  int getY() { return (int)Math.floor(dy); }
}
```

The compilation error can be avoided as shown: overriding is now correct, but…

Does the design error persist?

Not overriding, but overloading

```
public class Point {
  private int x = 0, y = 0, color;

  void move(int mx, int my) { x += mx; y += my; }


}

class RealPoint extends Point {
  double dx = 0.0, dy = 0.0;



  void move(double mx, double my) { dx += mx; dy += my; }



}
```

Method move(int,int) is inherited and coexists with the new method move(double,double) in RealPoint

There is no overwriting, but overloading of method in class RealPoint

Does move(int,int) make sense in RealPoint?

Overriding and overloading can coexist

```
public class Point {
  private int x = 0, y = 0, color;

  void move(int mx, int my) { x += mx; y += my; }


}

class RealPoint extends Point {
  double dx = 0.0, dy = 0.0;
  void move(int mx, int my) {
     move((double)mx, (double)my);
  }
  void move(double mx, double my) { dx += mx; dy += my; }



}
```

Method move(int,int) is overriden and overloaded with method move(double,double) in RealPoint

The complete example: well designed?

```
public class Point {
  private int x = 0, y = 0, color;

  public void move(int mx, int my) { x += mx; y += my; }
  public int getX() { return x; }
  public int getY() { return y; }
}

class RealPoint extends Point {
  double dx = 0.0, dy = 0.0;
  public void move(int mx, int my) {
     move((double)mx, (double)my);
   }
  public void move(double mx,double my) {dx +=mx; dy +=my;}

  public int getX() { return (int)Math.floor(dx); }
  public int getY() { return (int)Math.floor(dy); }
}
```

Does it make sense for RealPoint to inherit variables `x` and `y` of type `int`, if we add `dx` and `dy` of type `double`?

Example with attribute hiding

```
public class Point {
  private int x = 0, y = 0, color;

  public void move(int mx, int my) { x += mx; y += my; }
  public int getX() { return x; }
  public int getY() { return y; }
}

class RealPoint extends Point {
  double x = 0.0, y = 0.0;
  public void move(int mx, int my) {
     move((double)mx, (double)my);
   }
  public void move(double mx,double my) {x += mx; y += my;}

  public int getX() { return (int)Math.floor(x); }
  public int getY() { return (int)Math.floor(y); }
}
```

We can hide the inheritance of attributes x and y of type int, adding attributes with same name of type `double` in RealPoint

Attribute hiding

- Hiding of inherited attributes
- The definition in the subclass hides the one in the superclass
- The superclass is accessible from the subclass with **`super`**
- Attribute hiding
  - Both variables (in the superclass and subclass) coexist
  - The types of each variable can be different
  - A memory space is reserved for both definitions
- In this case, static binding is used
- In general, it is better to avoid attribute hiding (it is not as useful as method overriding)

Attribute hiding

```
public class Musician extends Person {
    String name; // hides attribute name in Person
    public void showNames() {
        System.out.println("Musician:  " + name);
        System.out.println("Person: " + super.name);
    }
}

// main block
Musician m = new Musician();
Person   p = m;
                            // access with static binding
m.name = "Stevie Wonder";   // Musician name
p.name = "Stevland Morris"; // Person name

((Musician)p).name = "Stevie Wonder";   // Musician name
((Person)m).name = "Stevland Morris"; // Person name
```

Exercise: What does this program print?

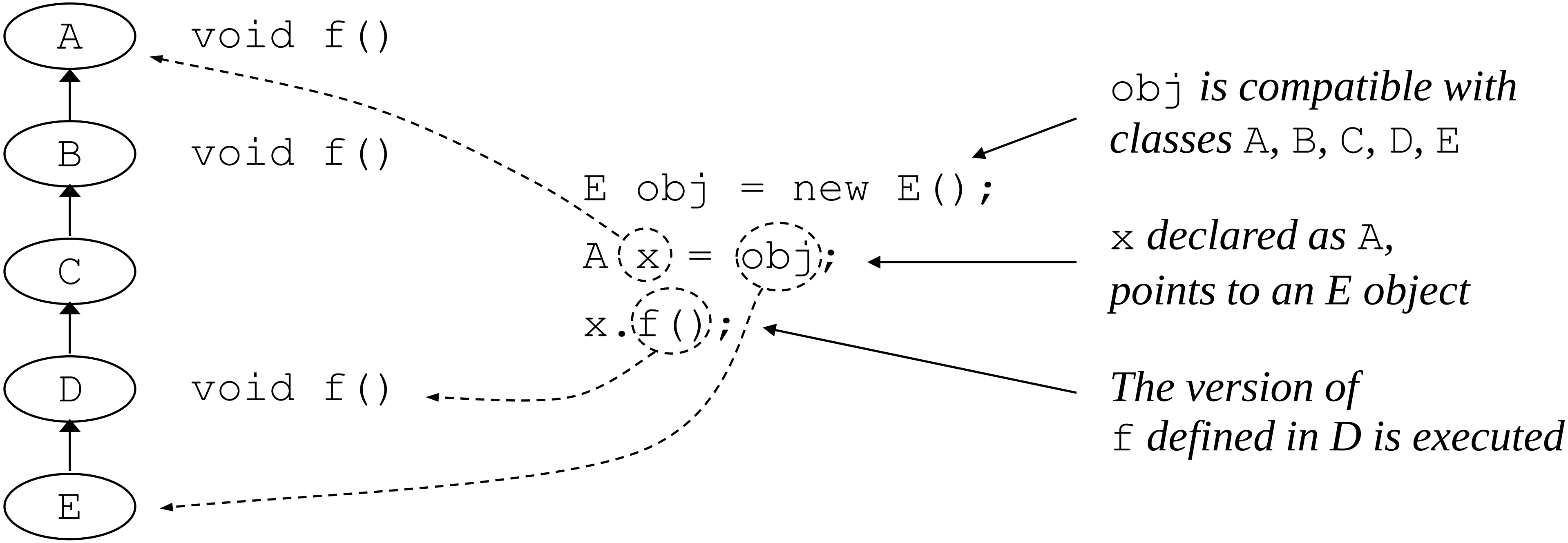
```
class AClass{
  public int a = 3;
}

class BClass extends AClass{
  public double a = 4.5;
}

public class Test2 {
  public static void main(String[] args) {
    BClass aa = new BClass();
    AClass ac = new AClass();
    AClass ab = new BClass();
    System.out.println(aa.a+" "+ac.a+" "+ab.a);
  }
}
```

Exercise: What does this program print?

```java
class AClass{
  public int a = 3;
}

class BClass extends AClass{
  public double a = 4.5;
}

public class Test2 {
  public static void main(String[] args) {
    BClass aa = new BClass();
    AClass ac = new AClass();
    AClass ab = new BClass();
    System.out.println(aa.a+" "+ac.a+" "+((BClass)ab).a);
  }
}
```

Dynamic binding

- Method overriding is resolved by dynamic binding at runtime
- The method definition of the most specific class of the object is executed, regardless of how the reference to the object has been declared



```
E obj = new E();
A x = obj;
x.f();
```

obj *is compatible with classes* A, B, C, D, E

x *declared as* A, *points to an E object*

The version of f *defined in D is executed*

- Static methods (class methods) have static binding

Dynamic binding: example

```
public class Person {
    String name;
    int age;
    public String toString() {
        return "name: " + name + "\nage: " + age;
    }
}

public class Employee extends Person {
    long grossSalary;
    Manager boss;
    public String toString() {
        return "name: " + name + "\nage: " + age +
               "\nSalary: " + grossSalary + "\nboss: " +
               ((boss == null)? name : boss.name);
    }
}
```

Dynamic binding: example(cont.)

```
public class Manager extends Employee {
    long incentive;
    ArrayList<Employee> team = new ArrayList<Employee>();
    public String toString() {
        return "name: " + name + "\nage: " + age +
               "\nSalary: " + grossSalary + "\nboss: " +
               ((boss == null)? name : boss.name) +
               "\nincentive: " + incentive;
    }
    public void setIncentive(long c) { incentive = c; }
}
```

How to improve the example?

Dynamic binding: example (cont.)

```
// main block
Manager  dir = new Manager();
Employee emp = new Employee();
Employee e   = dir;
Person p = new Person();
Person x = emp;
Person y = e;
String s;
s = p.toString();    // toString of Person
s = emp.toString();  // toString of Employee
s = dir.toString();  // toString of Manager
s = x.toString();    // toString of Employee
s = y.toString();    // toString of Manager
s = e.toString();    // toString of Manager
y.setIncentive(1500); // ERROR
```

The binding of arguments is static

It is not overriding but overloading

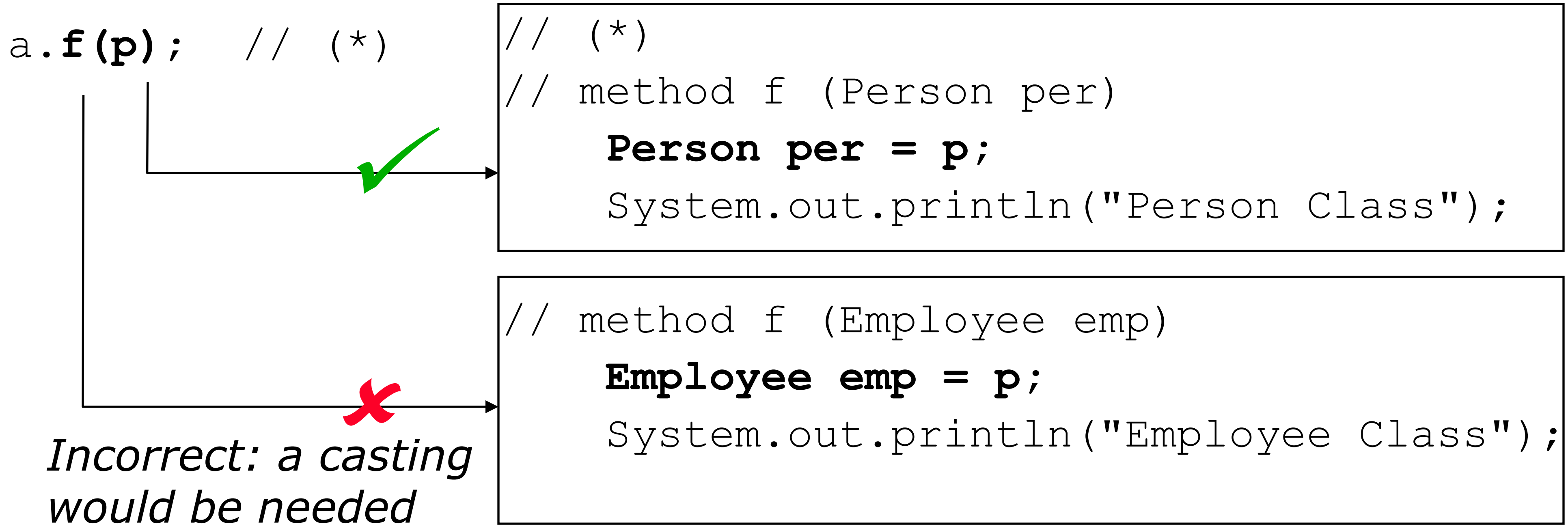
```
public class ClassA {
    public void f(Person per) {
        System.out.println("Person Class");
    }
    public void f(Employee emp) {
        System.out.println("Employee Class");
    }
}
```

The most specific compatible definition is executed

```
// main block
ClassA  a   = new ClassA();
Manager dir = new Manager();
Person  p   = dir;
a.f(dir);
a.f(p);  // (*)
Object x = p;
a.f(x); // ERROR
```

The binding of attributes is static. Why?

```
// main block
ClassA  a   = new ClassA();
Manager dir = new Manager();
Person  p   = dir;
```



```
a.f(p);  // (*)
```

```
// (*)
// method f (Person per)
    Person per = p;
    System.out.println("Person Class");
```

```
// method f (Employee emp)
    Employee emp = p;
    System.out.println("Employee Class");
```

Incorrect: a casting would be needed

When invoking a method, there is an assignment of actual to formal parameters

Class hierarchies and constructors

Constructors are not inherited nor can be overriden

Every (sub/super)class has its own constructor

- When creating an `Employee`, the constructor of `Person` should be invoked
- Automatic, implicit invocation to:
  - – The zero-parameter constructor of the superclass
  - – If not defined, an error is raised
- Otherwise, explicit invocation

  `super(…)` as the first instruction of the Employee constructor
- Invocation to other constructors of the same class: **`this(…)`**

Subclasses and constructors: example

```
public class Person {
    String name;
    int age;
    public Person(String str, int i) {
        name = str;
        age = i;
    }
    public String toString() {  ...  }
}
```

Error when creating an Employee:

The default constructor of **`Employee()`** invokes the default constructor of **`Person()`** that **now is undefined**

Before it was defined, because there was no constructor in **`Person`**

We can add one explicitly

```
    public Person() { name = ""; age = 0; }
```

Or better, add a new constructor to **`Employee`**

Subclasses and constructors: example

```
class Employee extends Person {
    long grossSalary;
    Manager boss;
    public Employee(String str, int i, long sueldo,
                    Manager dir) {
        name = str;     }
        age = i;        }  →  super(str, i);
        grossSalary = sueldo;
        boss = dir;
    }
    String toString() {  ...  }
}
```

Error: Even if we assign a value to **`name`** and **`age`**, we are still automatically invoking constructor `Person()`, which is not defined

Error when creating a Manager: the default constructor `Manager()` invokes the default constructor `Employee()` that is no longer defined

Subclasses and constructors: example

```
class Employee extends Person {
    long grossSalary;
    Manager boss;
    public Employee (String str, int i, long sueldo,
                     Manager dir) {
        super (str, i);
        grossSalary = sueldo;
        boss = dir;
    }
    String toString () {
        ...
    }
}
```

Always the first instruction

Subclasses and constructors: example

```
class Manager extends Employee {
   long incentive;
   ArrayList<Employee> team
                              = new ArrayList<Employee>();
   public Manager(String name, int age,
                       long sueldo, long incentive) {
       this(name, age, sueldo, null, incentive);
   }
   public Manager(String name, int age,
                       long sueldo, Manager boss,
                       long incentive) {
       super(name, age, sueldo, boss);
       this.incentive = incentive;
   }
   String toString() {
       ...
   }
   void setIncentive(long c) { incentive = c; }
}
```

Always the first instruction

Abstract classes, abstract methods

- Abstract class
  - We cannot create objects of abstract classes

    `new Person()` → Error if Person is **`abstract`**
  - Useful to define subclasses, providing them with attributes and methods
  - It can contain abstract and non-abstract methods
- Abstract methods
  - Methods with no code, they are declared but have no body
  - The body should be defined in every concrete subclass (by method overriding)
- Every class with an abstract method should be declared abstract
- Every subclass not overriding an abstract method should be declared abstract

Abstract class and subclasses: example

```
public abstract class Figure {
  public abstract double perimeter();

}

class Circle extends Figure {
  Point2D centre;
  double radius;
  public double perimeter() { return 2 * Math.PI * radius; }
}


class Triangle extends Figure {
  Point2D a, b, c;
  public double perimeter() {
    return a.distance(b) + b.distance(c) + c.distance(a);
  }
}
```

`perimeter()` *is an abstract method, without body, indicated with* **;**

These concrete subclasses need to override method `perimeter()`

Abstract class and subclasses: example

```
public abstract class Figure {
  public abstract double perimeter();
  public abstract void highlight();
}

abstract class Circle extends Figure {
  Point2D center;
  double radius;
  public double perimeter() { return 2 * Math.PI * radius; }
}

abstract class Triangle extends Figure {
  Point2D a, b, c;
  public double perimeter() {
    return a.distance(b) + b.distance(c) + c.distance(a);
  }
}
```

`perimeter()` *and* `highlight()` *are abstract methods*

`perimeter()` *is overriden, but not* `highlight()`*. The classes should remain abstract*

Abstract classes: extended example

```
public abstract class Figure {
  public abstract double perimeter();
  public abstract void highlight();
  // public abstract String toString(); would be wrong
}

abstract class FigureColor extends Figure { //Error w/o abstract
  Color lineColor, bgColor;
}

class Circle extends Figure {
  Point2D center;
  double radius;
  public double perimeter() { return 2 * Math.PI * radius; }
  public void highlight() { return; } // equiv to { }
  public String toString() {
    return "CIRC: center in” + center + " radius“ + radius;
  }
}
```

This class should be abstract since it does not override the inherited abstract methods

Even though the body is empty, `highlight()` *is overriden*

Abstract classes: extended example

```
public abstract class Figure {
  public abstract double perimeter();
  public abstract void highlight();
}

abstract class FigureColor extends Figure { //Error w/o abstract
  Color lineColor, bgColor;
}

class TriangleColor extends FigureColor {
  Point2D a, b, c;
  public double perimeter() {
    return a.distance(b) + b.distance(c) + c.distance(a);
  }
  public void highlight() {
    lineColor.highlight();
    bgColor.highlight();
  }
}
```

Abstract classes: extended example

```
public abstract class Figure {
  public abstract double perimeter();
  public abstract void highlight();
}

abstract class FigureColor extends Figure { //Error w/o abstract
  Color lineColor, bgColor;
}

class TriangleColor extends FigureColor {
  Point2D a, b, c;
  public double perimeter() {
    return a.distance(b) + b.distance(c) + c.distance(a);
  }
  public void highlight() {
    lineColor.highlight();
    bgColor.highlight();
  }
}
```

Can a private method be abstract?

Usefulness of abstract methods

```
public class FigureGroup { // Container of Figure objects
  private ArrayList<Figure> figures
                                = new ArrayList<Figure>();

  public void addFigure(Figure fig) {
    figures.add(fig);
  }

  public void highlight() {
    Iterator<Figure> iter = figures.iterator();
    while (iter.hasNext())
       iter.next().highlight();
  }
}
```

Instantiating the generic class `ArrayList<T>` *with* `Figure` *as* `T` *we can add objects of any subclass of* `Figure` *to the list*

We can invoke `highlight()` *over every* `Figure` *without needing to know at compile time, the subclass of* `Figure` *holding the method body*

Usefulness of abstract methods

```
public class FigureGroup {// Container of Figure objects
  private List<Figure> figures = new ArrayList<Figure>();


  public void addFigure(Figure fig) {
    figures.add(fig);
  }

  public void highlight() {

    for (Figure f: figures) {
       f.highlight();
    }
  }
}
```

Also valid: List is an interface, implemented by collections like ArrayList or Vector.

We can use the improved for loop to iterate over collections

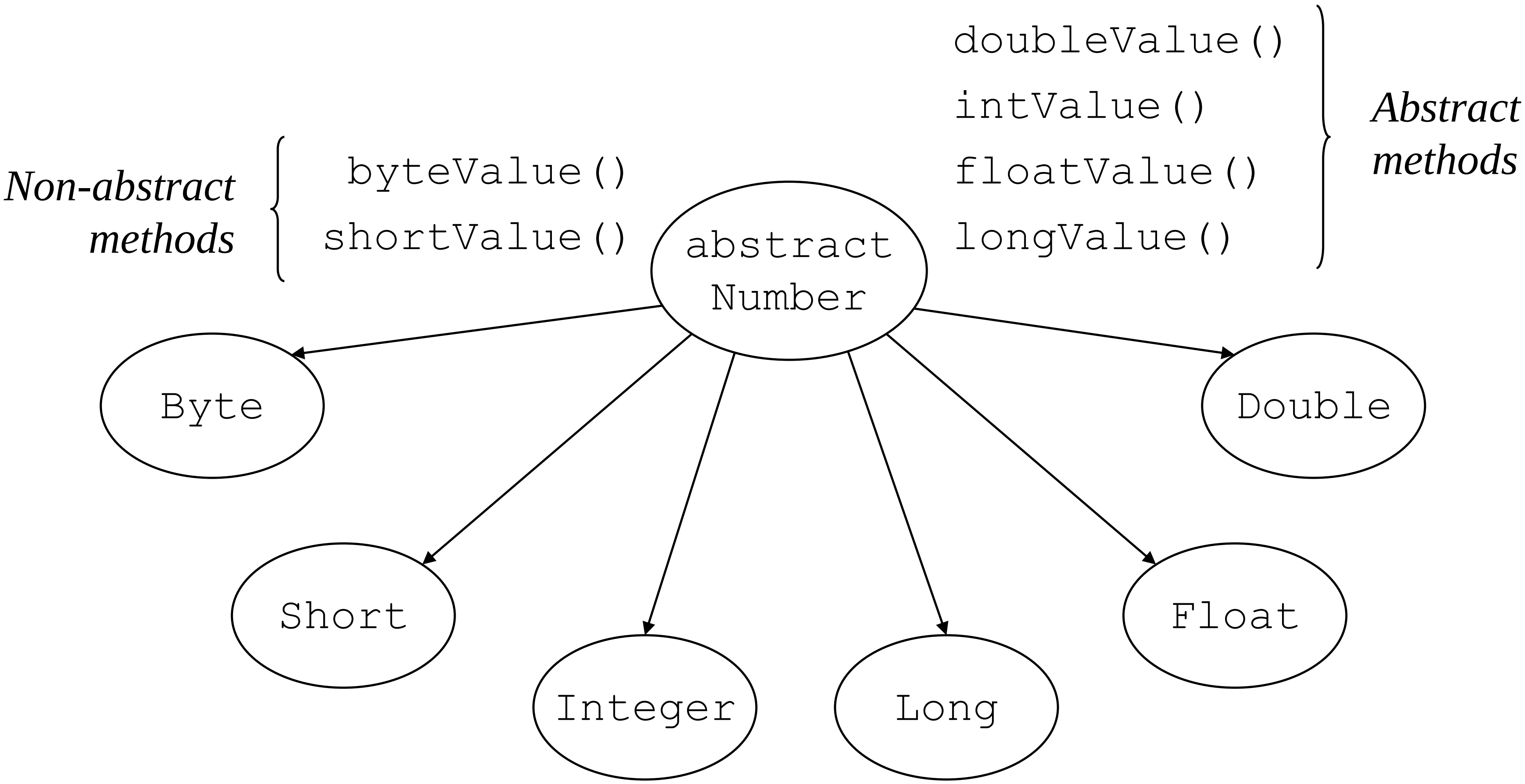
Exercise

- Build a program that:
  - Emulates a simple file system.
  - A file has a name, size in bytes, and a type (R, RW, W)
  - A folder has a name and may contain files or folders. Its size is given by the size of the files and folders it contains.

- Improve the program to avoid adding an element e to a folder d if:
  - e is already contained in d, or in any subfolder.
  - d is contained in e, or in any subfolder.

How would you check that an Element is not inside two different folders?

`java.lang.Number` is an abstract class



`final` modifier on variable, method and class

- Variables with **`final`** (*similar to constants*)
  - The first assigned value cannot be changed afterwards
  - Instance attributes with **`final`** should
    be initialized in its declaration, or
    be assigned in every constructor
  - Class attributes with **`final`** modifier should
    be initialized in its declaration, or
    be assigned in the class initializer
- Methods with **`final`**
  - Cannot be overriden in subclasses
- Classes with **`final`**
  - Cannot be *extended* with subclasses

Classes, methods and variables with `final`: example

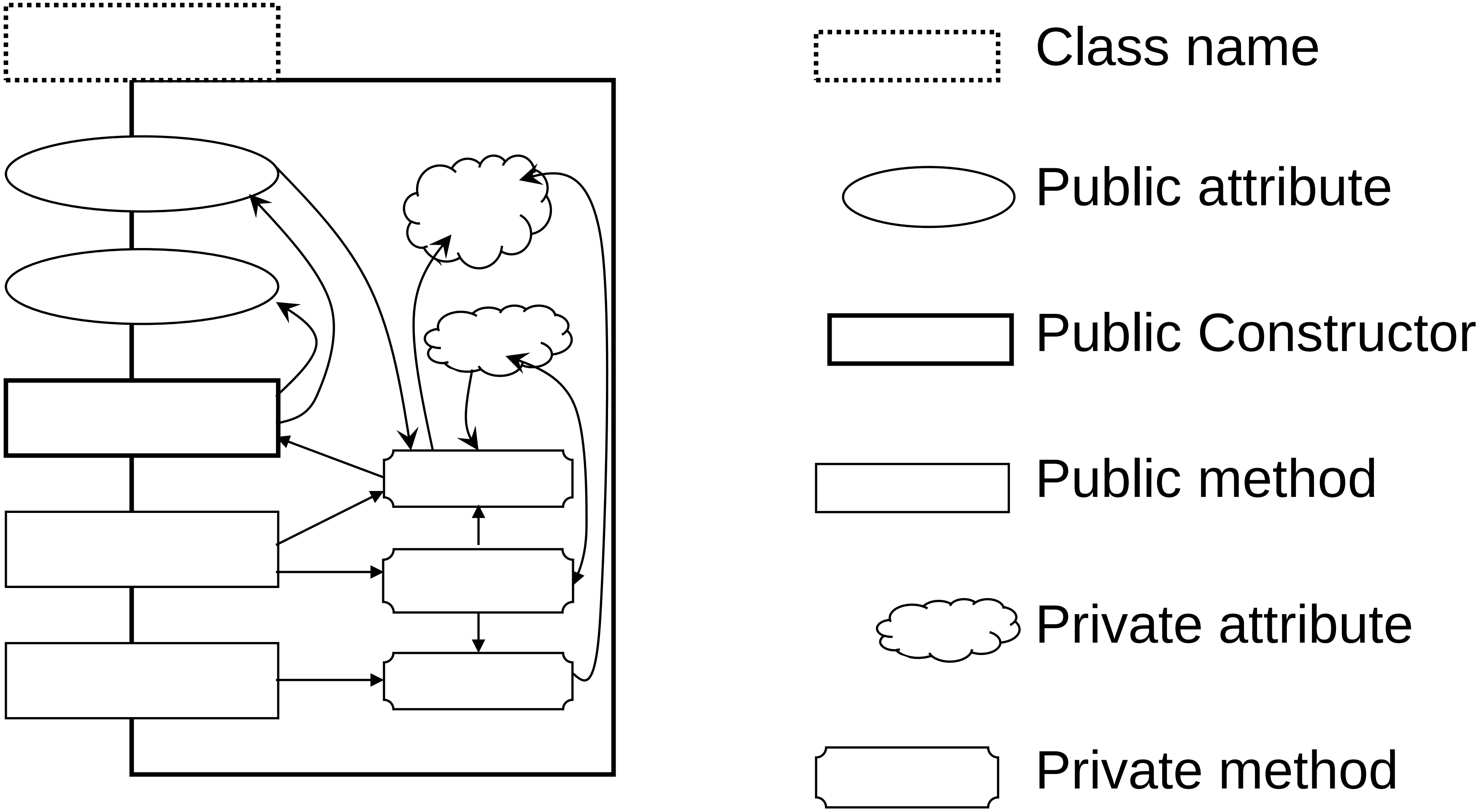
```
final class ClassA {
  ...
}
class ClassB extends ClassA { // Error: ClassA forbids subclasses
   private final int x;       // Error: x  is not initialized
   private final int y = 0;   // OK: y  is initialized
   private final int z;       // OK: z  not initialized but …
                              // initialized in every constructor
   public ClassB() { x = 0; z = 0; }
   public ClassB(int n) { z = n; }

   public final double f(int x) { return (x-1)/(x+1); }
}
class ClassC extends ClassB {
   public double f(int x) { //Error: f  cannot be overriden
      return (x-2)/(x+2);
   }
}
```

Notion of package in Java

- Set of related classes, offered to the programmer as a closed software unit
- Only the public classes within the package are accessible from outside (using **`import`** or with **`package.class`**)
- Avoids naming conflicts between classes of different packages
- Every class can only belong to one package
- Packages can be divided hierarchically in subpackages
- If no package is defined for a class, it is included in the default package (without name, better not to use, as it cannot be imported)
- In addition to classes, packages can contain interfaces and enums

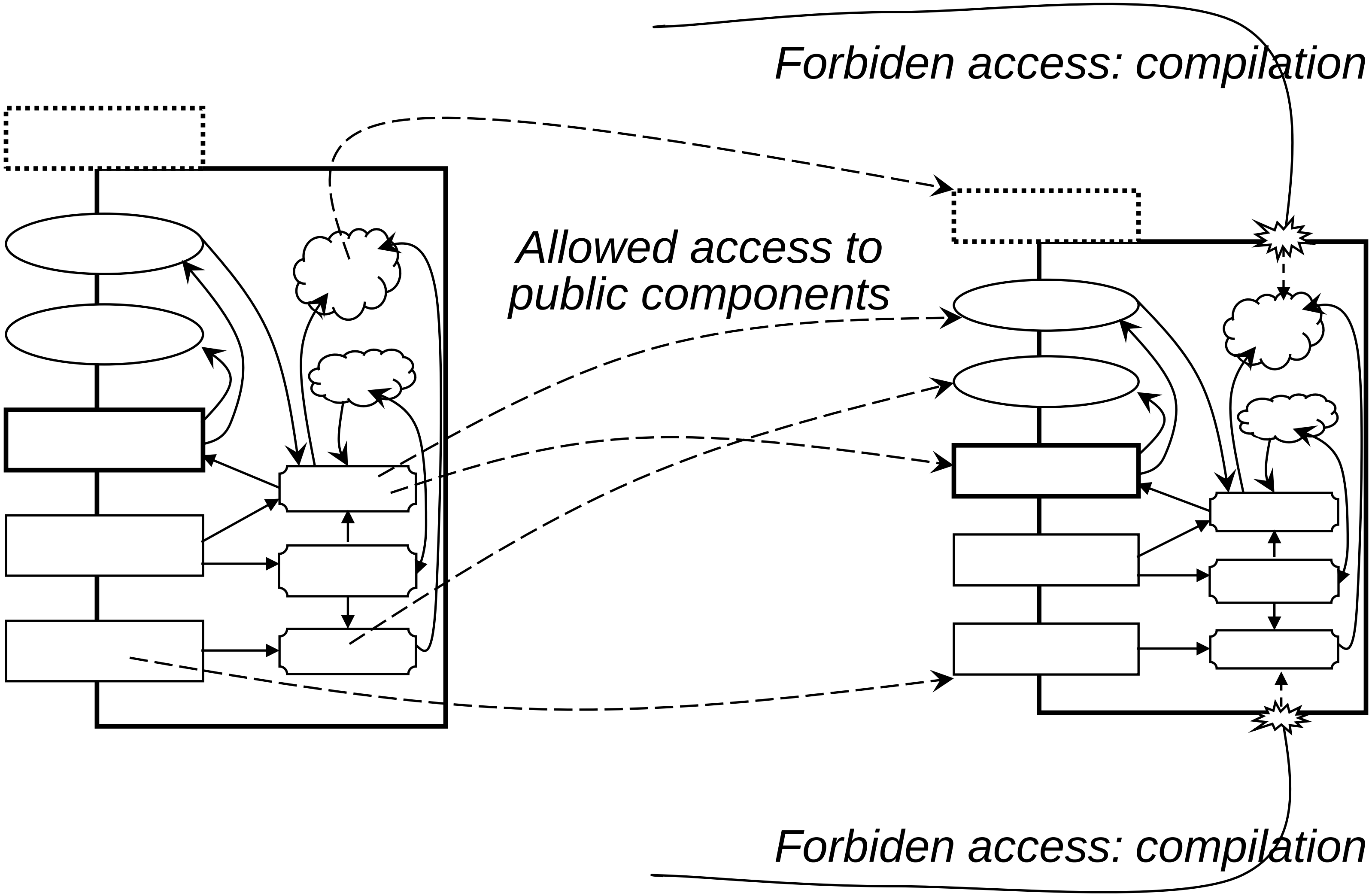
(Simplified) schema of classes

Legend:



- Class name
- Public attribute
- Public Constructor
- Public method
- Private attribute
- Private method

Relations and accesses between separate classes



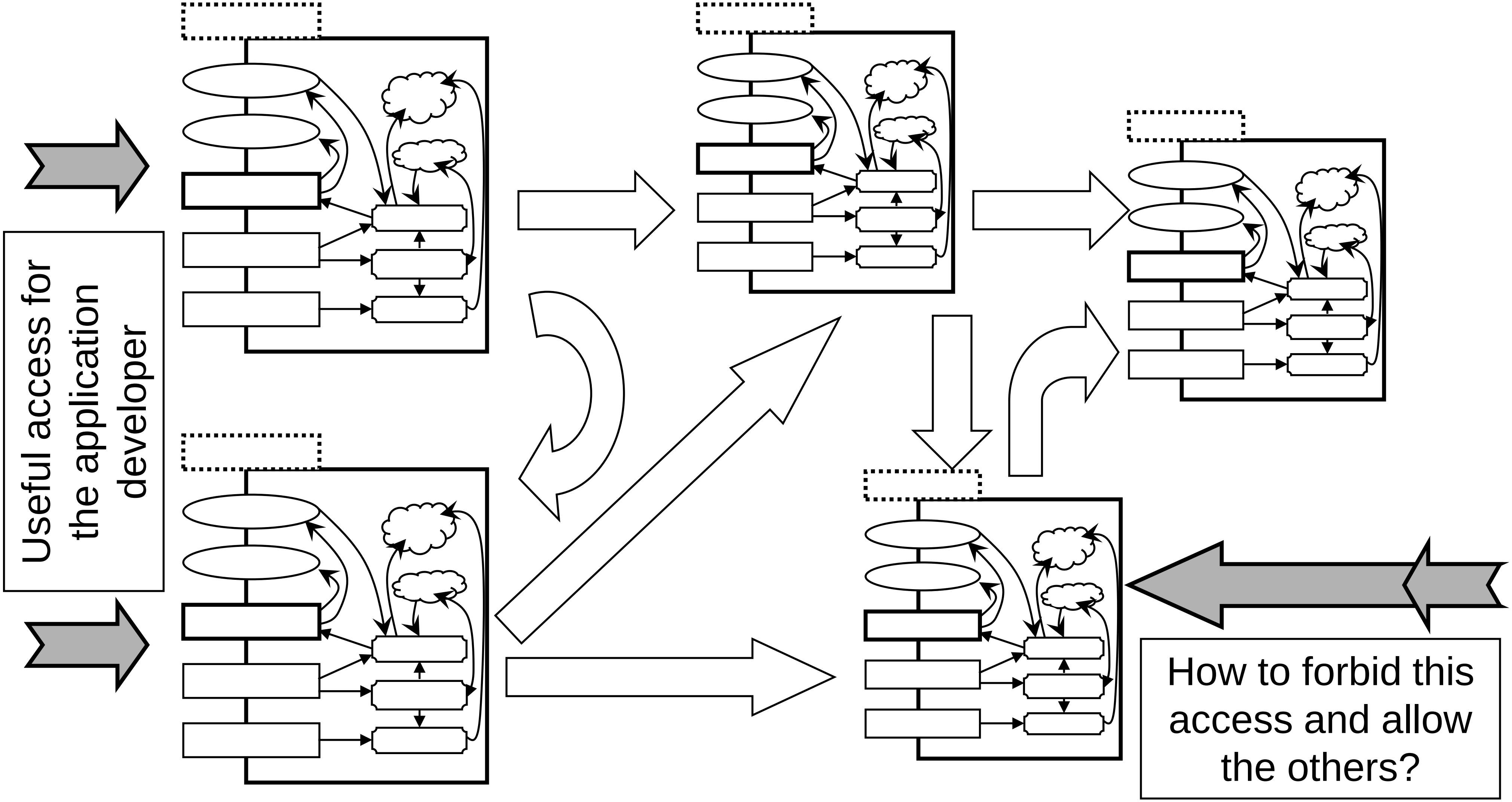
Forbiden access: compilation error

Allowed access to public components

Forbiden access: compilation error

Interrelated classes

If only some classes make sense for the developer, we should package them and define an interface

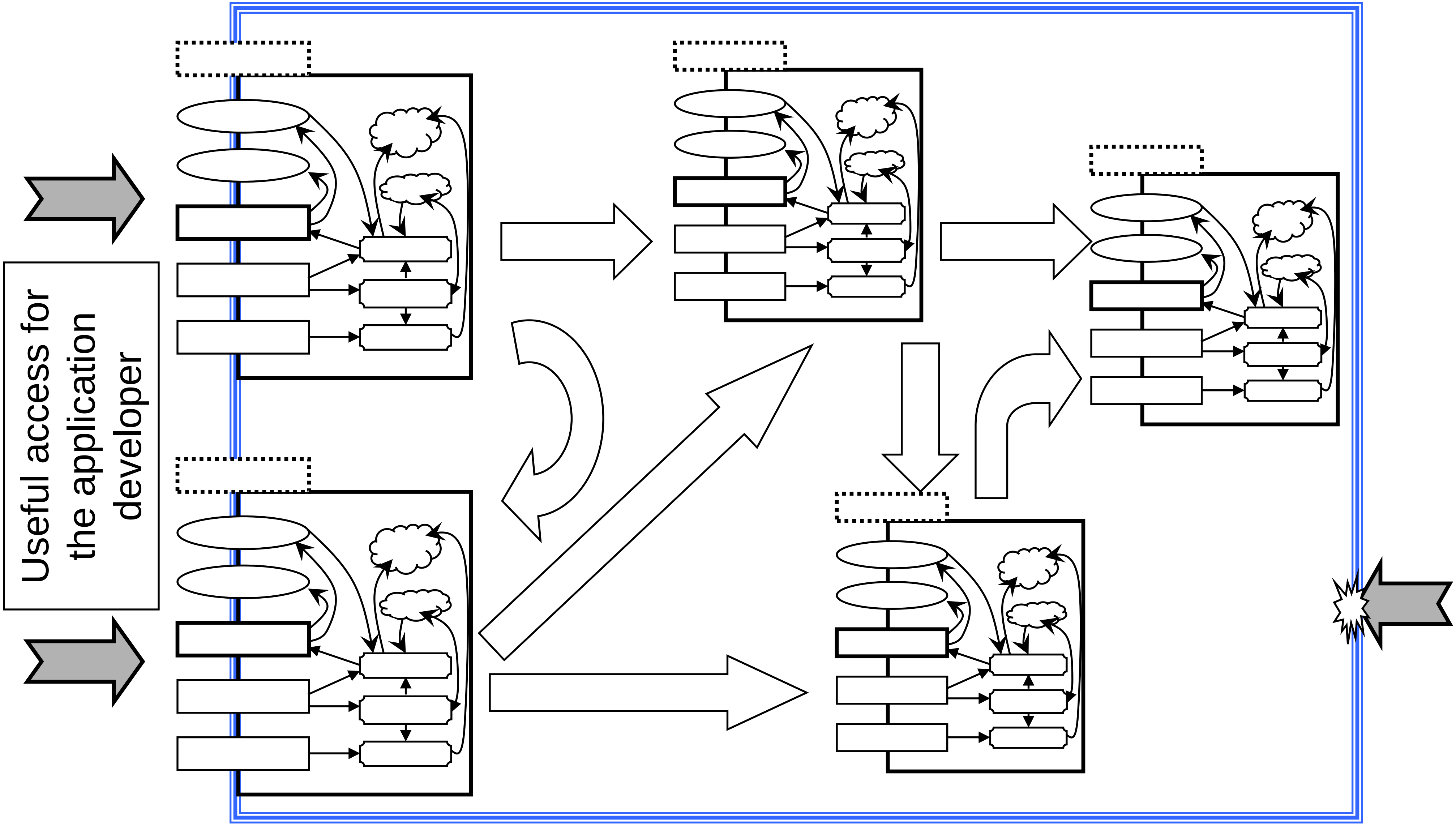


Useful access for the application developer

How to forbid this access and allow the others?

Classes grouped within a package

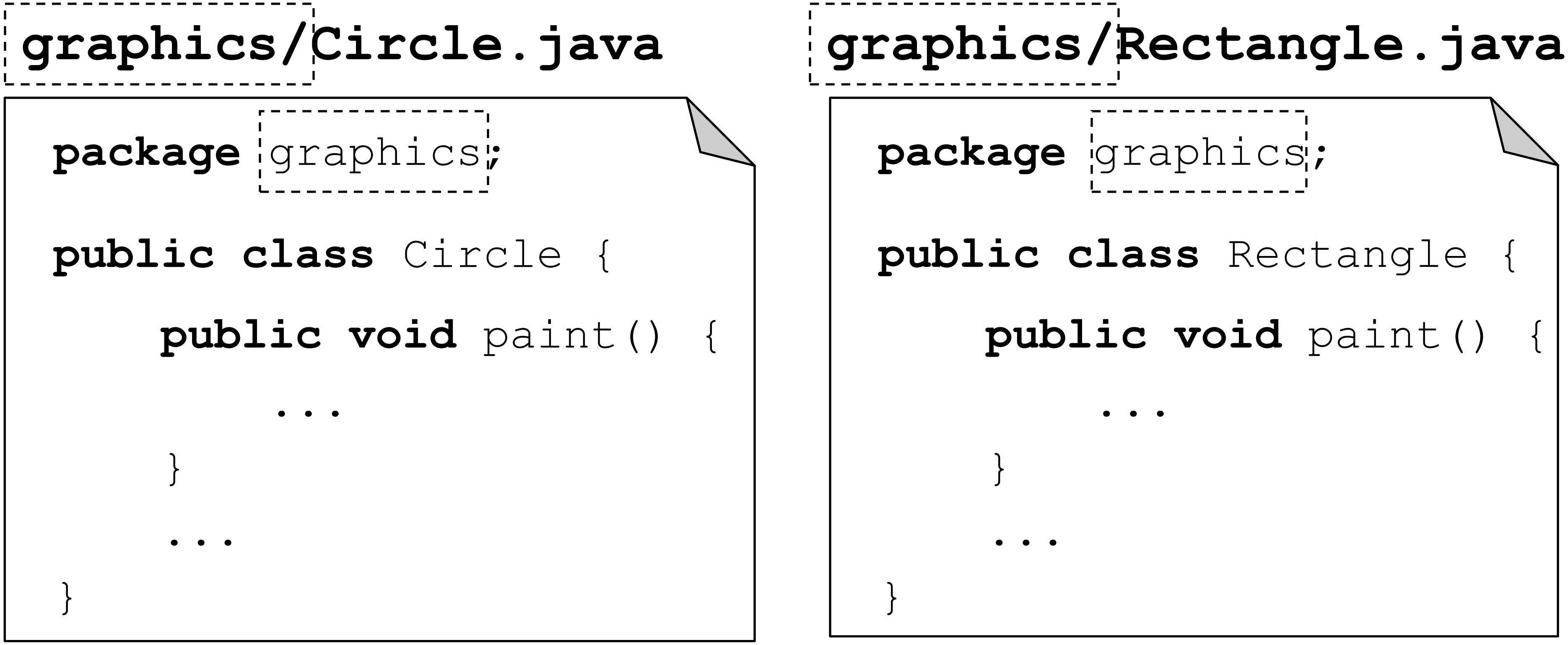
Only the public classes are accessible



How to define packages

Starting each compilation unit with a **`package`** declaration

Storing all compilation units of a same package in a folder with same name as the package



graphics/Circle.java

```
package graphics;

public class Circle {

    public void paint() {

        ...

    }

    ...

}
```

graphics/Rectangle.java

```
package graphics;

public class Rectangle {

    public void paint() {

        ...

    }

    ...

}
```

How to use classes of another package

Direct use with the notation **package.class**

```
...
graphics.Circle c = new graphics.Circle();
c.paint ();
...
```

Import one class

```
import graphics.Circle;
...
Circle c = new Circle();
c.paint();
...
```

Import all classes within a package

```
import graphics.*;
...
Circle    c = new Circle();
Rectangle r = new Rectangle();
c.paint(); r.paint();
...
```

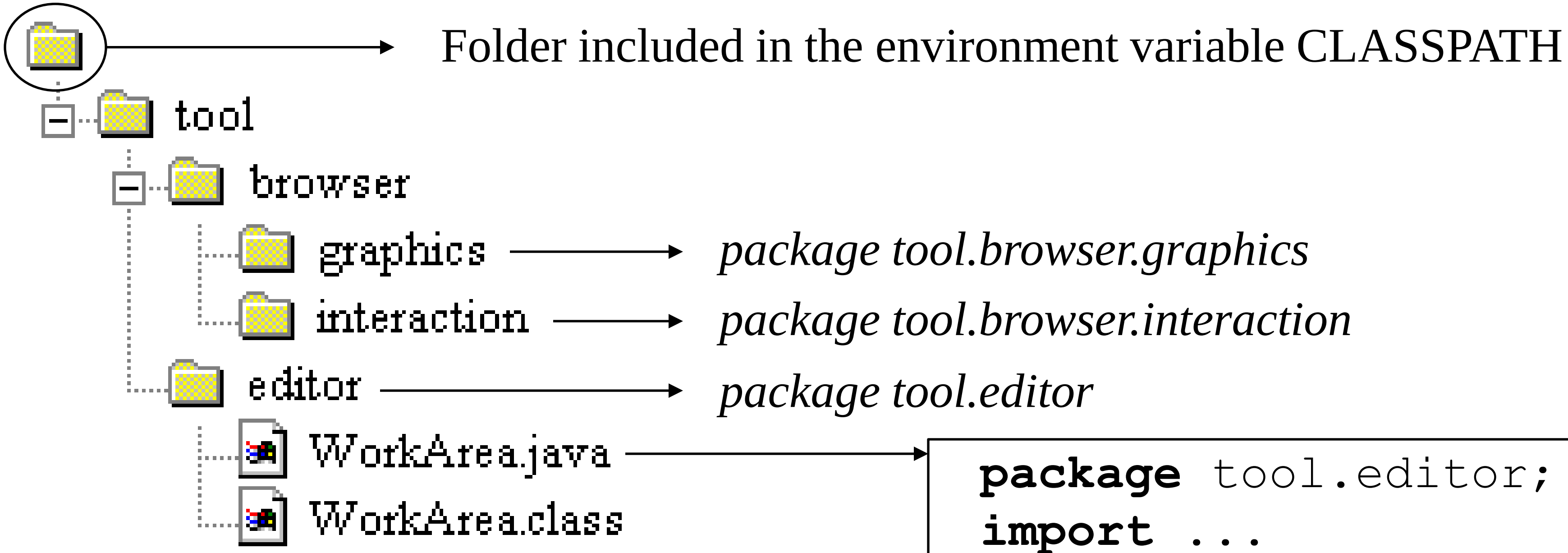
Import static

- Import a static attribute or method declared in another class
- Avoids having to use `<name-class>.<name-method>()`, instead `<name-method>()` can be used
- Example:

```
import static java.lang.Math.PI;
import static java.lang.Math.abs;

public class Main {
   public static void main(String ...args) {
      System.out.println("PI="+PI);
      System.out.println("abs(-45)="+abs(-45));
  }
}
```

Packages in folders

- Package name → folder structure
- Environment variable **`CLASSPATH`** in the operating system: contains a list of the folders where Java will look for packages



Folder included in the environment variable CLASSPATH

package tool.browser.graphics

package tool.browser.interaction

package tool.editor

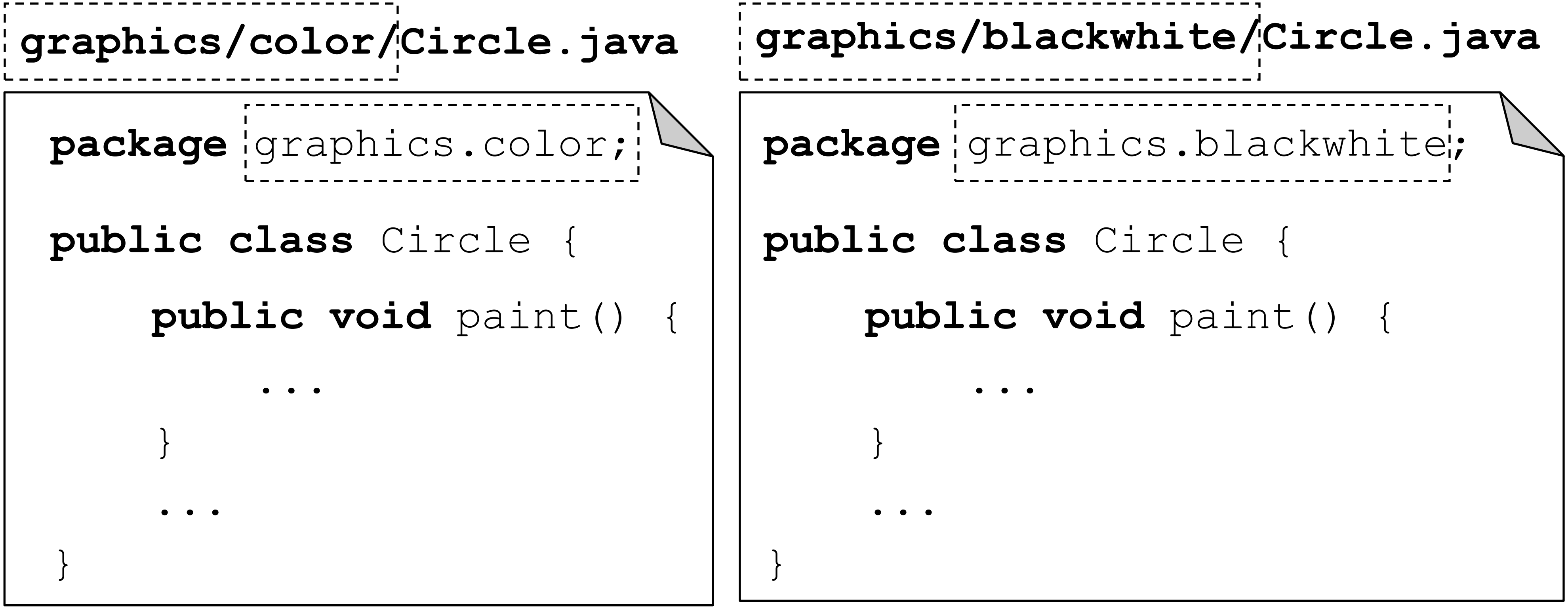
```
package tool.editor;
import ...

public class WorkArea {
      ...
}
```

- Automatically imported packages:
  - **`java.lang`**
  - Current package

How to define <u>sub</u>packages

Start each compilation unit with a **`package`** declaration

Place the compilation units of a same subpackage in a subfolder with same name, and respecting the hierarchy of folders/packages



graphics/color/Circle.java

```
package graphics.color;

public class Circle {

    public void paint() {

        ...

    }

    ...

}
```

graphics/blackwhite/Circle.java

```
package graphics.blackwhite;

public class Circle {

    public void paint() {

        ...

    }

    ...

}
```

Predefined packages in Java (API)

java.applet
java.awt
java.awt.datatransfer
java.awt.event
java.awt.image
java.beans
java.io
java.lang
java.lang.reflect
java.math
java.net

java.rmi
java.rmi.dgc
java.rmi.registry
java.rmi.server
java.security
java.security.acl
java.security.interfaces
java.sql
java.text
java.util
java.util.zip
…

http://docs.oracle.com/javase/7/docs/api/

(*Close to 300*)

Access control/visibility:

We have already used `public` and `private`

```
class ClassA {
  public int x;
  private int y;
  public void f() { ... }
  private void g() { ... }
  void h()
    x = 2;
    y = 6;
    f();
    g();
    ClassA a = new ClassA();
    a.x = 2;
    a.y = 6;
    a.f();
    a.g();
  }
}
```

```
class ClassB {
  void m() {
    ClassA a = new ClassA();
    a.x = 2;
    a.y = 6; //Error private
    a.f();
    a.g();   //Error private
    a.h();
  }
}
```

What’s the visibility of `h()`?

Access control: other possibilities

Visibility possibilities for attributes, methods and constructors

	Class	Package	Subclass	Any
`private`	X			
(default: *package)*	X	X		
`protected`	X	X	X	
`public`	X	X	X	X

Control access for classes:

- *top-level* classes, only **`public`** or *package* (default)
- *internal classes*, also **`protected`** or **`private`**

***Remark**: protected and package are equivalent if the superclass and the subclass are in the same package*

Access control for attributes and methods

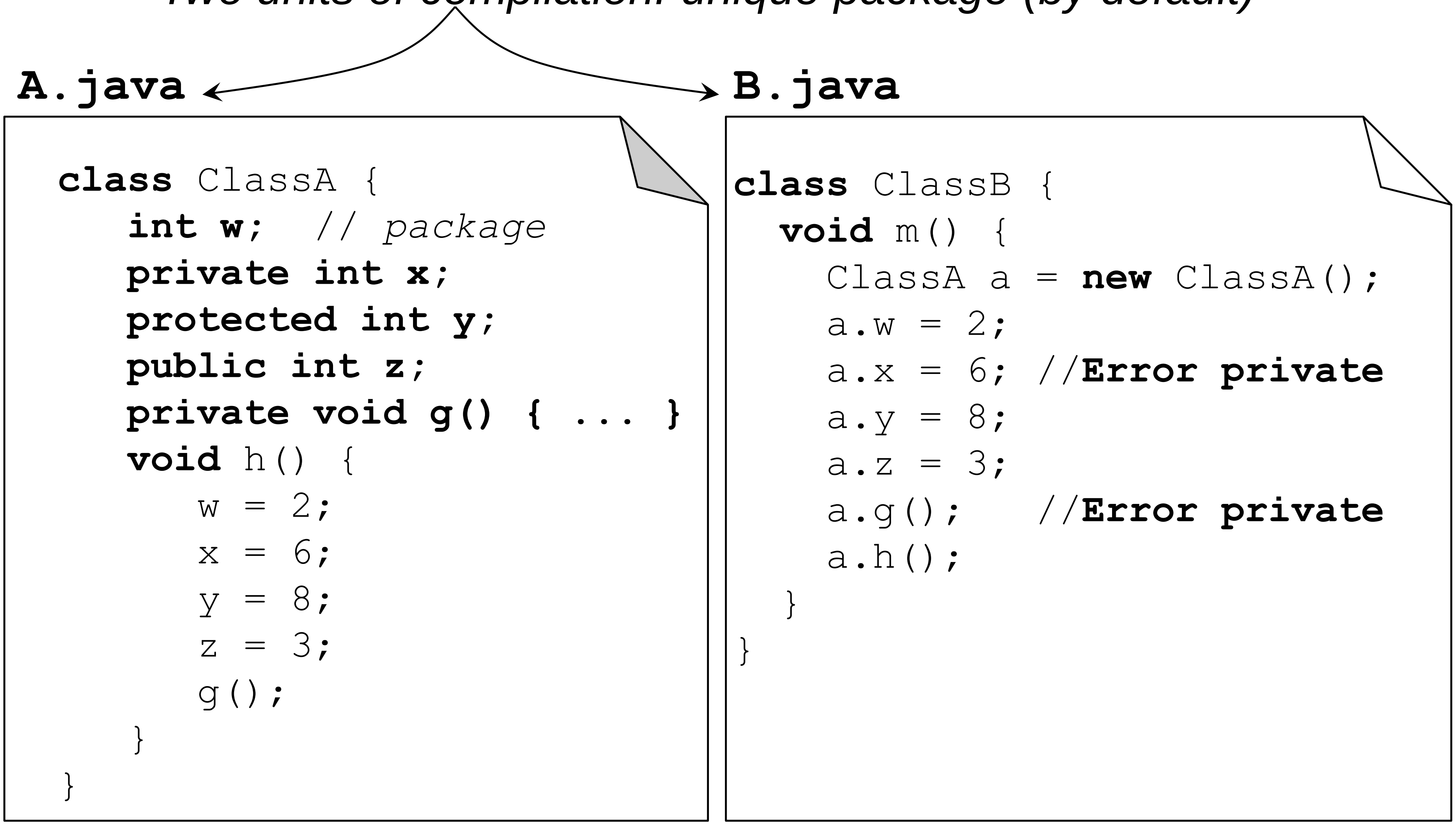
archive.java ↔ *Unique unit of compilation*

```
class ClassA {
    int w;  // package
    private int x;
    protected int y;
    public int z;
    private void g() { ... }
    void h() {
        w = 2;
        x = 6;
        y = 8;
        z = 3;
        g();
    }
}
```

```
class ClassB {
  void m() {
    ClassA a = new ClassA();
    a.w = 2;
    a.x = 6; //Error private
    a.y = 8;
    a.z = 3;
    a.g();   //Error private
    a.h();
  }
}
```

Access control for classes

Two units of compilation: unique package (by default)



A.java

```
class ClassA {
    int w;  // package
    private int x;
    protected int y;
    public int z;
    private void g() { ... }
    void h() {
        w = 2;
        x = 6;
        y = 8;
        z = 3;
        g();
    }
}
```

B.java

```
class ClassB {
  void m() {
    ClassA a = new ClassA();
    a.w = 2;
    a.x = 6; //Error private
    a.y = 8;
    a.z = 3;
    a.g();   //Error private
    a.h();
  }
}
```

Access control to classes within packages

Two compilation units and ***two packages***: **`p1`** and default

A.java

```
package p1;
class ClassA { // package
   int w;  // package
   private int x;
   protected int y;
   public int z;
   private void g() { ... }
   void h() {
      w = 2;
      x = 6;
      y = 8;
      z = 3;
      g ();
   }
}
```

B.java

```
class ClassB {
  void m() {
    ClassA a = new ClassA();
    // Error:
    // ClassA not found
    // import p1, or use full
    // name p1.ClassA()
    // Error:
    // ClassA not public in p1
  }
}
```

Access control to classes within packages

Two compilation units and two packages

p1/ClassA.java

```
package p1;
public class ClassA {
   int w;  // package
   private int x;
   protected int y;
   public int z;
   private void g() { ... }
   void h() {
      w = 2;
      x = 6;
      y = 8;
      z = 3;
      g ();
   }
}
```

B.java

```
import p1.ClassA;
class ClassB {
  void m() {
    ClassA a = new ClassA();
    a.w = 2; //Error package
    a.x = 6; //Error private
    a.y = 8; //Error protected
    a.z = 3;
    a.g();   //Error private
    a.h();   //Error package
  }
}
```

Also possible: `import p1.*;`

Or w/o import using `p1.ClassA()`

Access control to classes within packages

Two compilation units and **only one package***:* `p1`

p1/ClassA.java

```
package p1;
public class ClassA {
    int w;  // package
    private int x;
    protected int y;
    public int z;
    private void g() { ... }
    void h() {
        w = 2;
        x = 6;
        y = 8;
        z = 3;
        g ();
    }
}
```

p1/B.java

```
package p1;
class ClassB {
  void m() {
    ClassA a = new ClassA();
    a.w = 2; // ok package
    a.x = 6; //Error private
    a.y = 8; // ok protected
    a.z = 3;
    a.g();   //Error private
    a.h();   // ok package
  }
}
```

Classes in different packages

Two compilation units and ***two packages****:* `p1` and `p2`

p1/ClassA.java

```
package p1;
public class ClassA {
    int w;  // package
    private int x;
    protected int y;
    public int z;
    private void g() { ... }
    void h() {
        w = 2;
        x = 6;
        y = 8;
        z = 3;
        g ();
    }
}
```

p2/ClassB.java

```
package p2;
import p1.*;
public class ClassB {
  void m() {
    ClassA a = new ClassA();
    a.w = 2; //Error package
    a.x = 6; //Error private
    a.y = 8; //Error protected
    a.z = 3;
    a.g();   //Error private
    a.h();   //Error package
  }
}
```

Grouping classes within packages

Public classes within a package can be imported

p1/ClassA.java

```
 package p1;
public class ClassA {
    ...
 }

 // non-public class of
 // package p1
 class Aux {
    ...
 }


 }
```

p2/ClassB.java

```
package p2;
import p1.*;
public class ClassB {
  void m() {
    ClassA a = new ClassA();
    ...
    Aux u= new Aux(); // Error
    ...
  }
}
```

Even with `p1.Aux` there is an error

Enabling access to subclasses: `protected`

But only through expressions of type compatible with the subclasses

p1/ClassA.java

```
package p1;
public class ClassA {
   int w;  // package
   private int x;
   protected int y;
   public int z;
   private void g() { ... }
   void h() {
      w = 2;
      x = 6;
      y = 8;
      z = 3;
      f ();
   }
}
```

p2/ClassB.java

```
package p2;
import p1.*;
public class ClassB
             extends ClassA {
  void m() {
    ClassA a = new ClassA();
    ClassB b = new ClassB();

    a.y = 8; //Error protected
    b.y = 7; // ok protected
    a = new ClassB();
    a.y = 6; //Error protected
    y = 5;   // ok protected
  }
}
```

***Remark**: Only problematic if ClassA and ClassB are in different packages*

Example of access control: public, private, *package*

```
class Person {
    private String name;
    private int age;
    public String toString() {
        return "name: " + name + "\nage: " + age;
    }
}

class Employee extends Person {
    long grossSalary;
    Manager boss;
    public String toString() {
        return "name: " + name + "\nage: " + age +
               "\nSalary: " + grossSalary + "\nboss: " +
               ((boss == null)? name : boss.name);
    }
}
```

Error: name and age are private

Example of access control: protected (I)

```
class Person {
    protected String name;
    protected int age;
    public String toString() {
        return "name: " + name + "\nage: " + age;
    }
}

class Employee extends Person {
    long grossSalary;
    Manager boss;
    String toString() {
        return super.toString () +
               "\nSalary: " + grossSalary + "\nboss: " +
               ((boss == null)? name : boss.name);
    }
}
```

Error: can only be public

Correct even if Employee and Person in different package

In different package there would be ab error jf `boss` *was Person, but not if it was Employee*

Example of access control: protected (II)

Warning: `toString` *is inherited with public*

```
package staff;

public class Person {
    protected String name;
    protected int age;
    protected String idString () {
        return "name: " + name + "\nage: " + age;
    }
}
```

```
package staff;

...
// In any class
Person p = new Person ();
p.idString ();
...
```

```
package x;

...
// In any class
staff.Person p =
    new staff.Person();
p.idString (); // Error
...
```

Exercise of access control: protected (III)

```
package p1;

public class A {
   int w; // package
   private int x;
   protected int y;
   public int z;
}
```

```
package p2;

class C {
   void h() {
      p1.A a = new p1.A();
      a.w = 2;
      a.x = 6;
      a.y = 8;
      a.z = 3;
   }
}
class D extends p1.A {
   void h() {
      p1.A a = new p1.A();
      w = 2; a.w = 2;
      x = 2; a.x = 6;
      z = 3; a.z = 3;
             a.y = 8;
      y = 8;
      D d = new D ();
      d.y = 8;
   }
}
```