Programming Software Systems

Introduction to Programming
for the Computer Engineering Track

Lecture 8 + Tutorial 8
An Introduction to Java

Eugene Zouev
Fall Semester 2020
Innopolis University

What We Have Learnt

- **Classes** as program building blocks
- Class **instances**
- **Value types** and **reference types**
- **Access control**: public and private members
  Encapsulation
- **Constructors**
- `null` & `this`
- **Parameter passing**
- **Packages**

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:

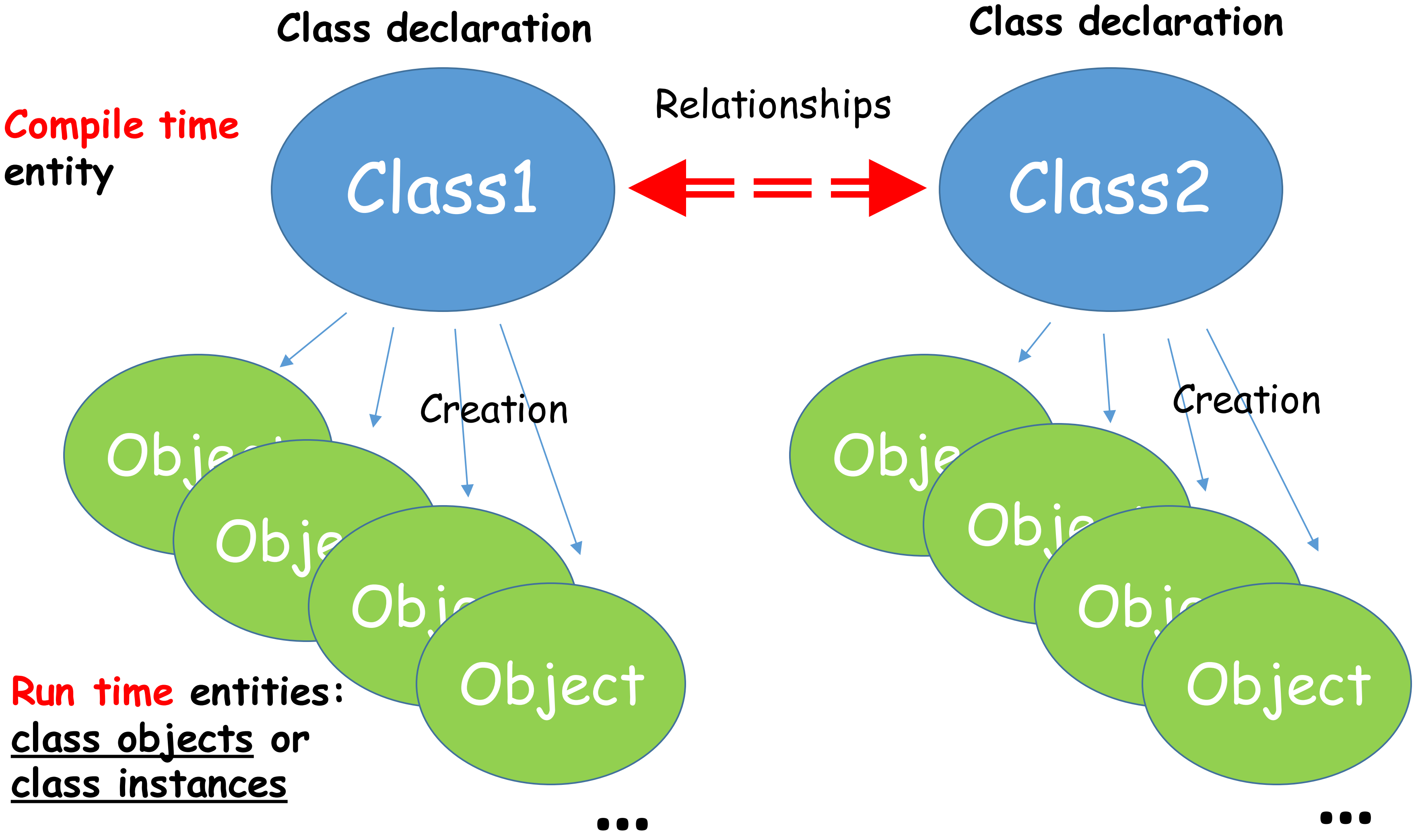
Class is a language construct comprising **algorithms** (in form of **functions**) and **data** the algorithms work on

Simplified!

Classes & Objects

From the previous lecture

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



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
```

What's For Today

- Method overloading
- **Inheritance**
- **Polymorphism**

Two other cornerstones of OOP

Encapsulation:
the first cornerstone of the object-oriented approach.

Method Overloading

The case with methods of the same name

```
class Class
{
(1) void f(int x) { ... }
(2) void f(float x) { ... }
(3) void f(C1 x) { ... }








}
```

Signature is a composition of the name of the function, the number of its parameters, and the types of its parameters.

Type of function's return value is not included to the signature

Multiple declarations for attributes are (of course) prohibited. However, **multiple declarations for methods are allowed if their signatures are different**.

Method Overloading

The case with methods of the same name

```
class Class
{
(1) void f(int x) { ... }
(2) void f(float x) { ... }
(3) void f(C1 x) { ... }

    void g()
    {
      f(1);           // calls (1)
      f(1.1);         // calls (2)
      f(new C1());    // calls (3)
      f(new C2());    // Error
      f(1,true);      // Error
    }
}
```

Multiple declarations for attributes are (of course) prohibited. However, **multiple declarations for methods are allowed <u>if their signatures are different</u>.**

Signature is a composition of the name of the function, the number of its parameters, and the types of its parameters.

Type of function's return value is not included to the signature

Method Overloading

The case with methods of the same name

```
class Class
{
(1) void f(int x) { ... }
(2) void f(float x) { ... }
(3) void f(C1 x) { ... }

    void g()
    {
      f(1);           // calls (1)
      f(1.1);         // calls (2)
      f(new C1());    // calls (3)
      f(new C2());    // Error
      f(1,true);      // Error
    }
}
```

Signature is a composition of the name of the function, the number of its parameters, and the types of its parameters.

Type of function's return value is not included to the signature

Multiple declarations for attributes are (of course) prohibited. However, **multiple declarations for methods are allowed if their signatures are different**.

Such a feature is called **method overloading**: there are several methods in the same class (i.e., in the same scope).

Sometimes the feature is called **compile-time (or static) polymorphism**. Compiler chooses which method to use analyzing types of arguments of the call.

Inheritance

The second cornerstone of object orientation (*after encapsulation*)

Another practically useful mechanism is **aggregation**

What Was Before: Taxonomy

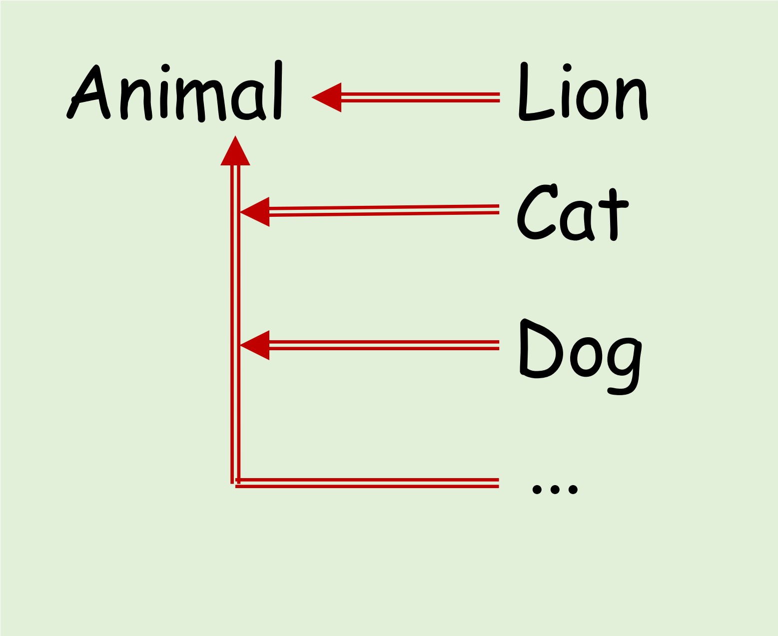
Taxonomy (classification) is one of the fundamental tools of science

- It enables us to keep the description of complex phenomena and concepts simple

=> Therefore, designing software we should first think about how to **classify** entities we are going to represent.

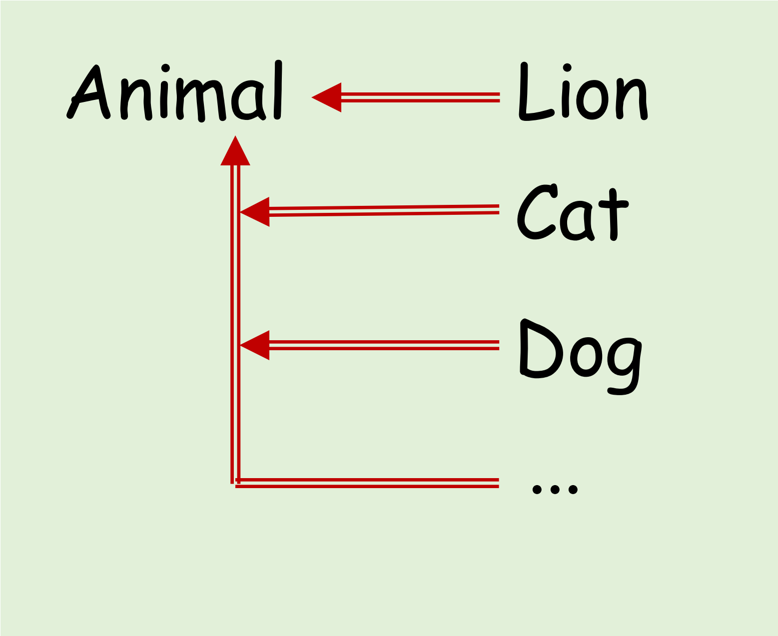
Inheritance 1

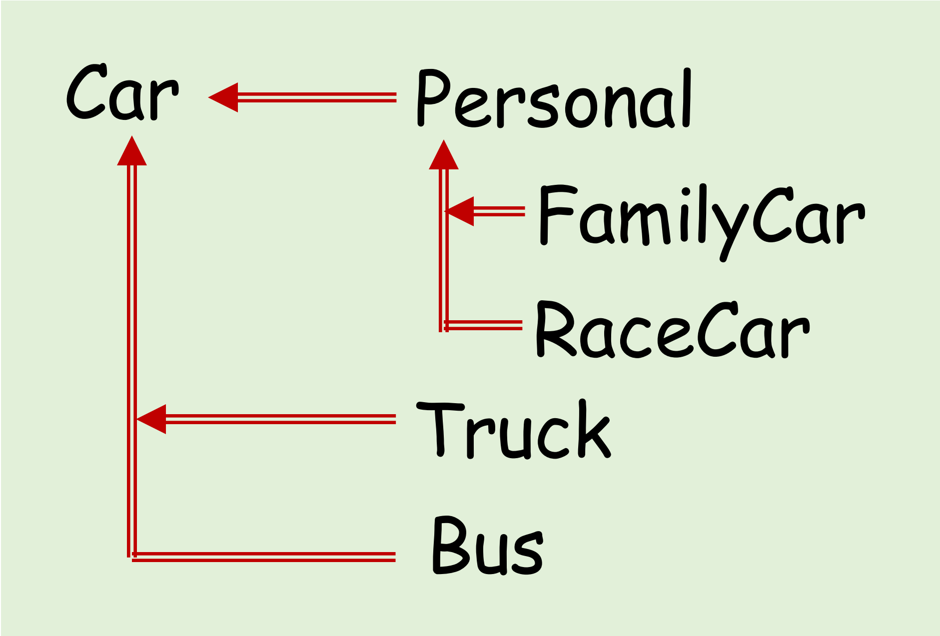
Define new entities based on existing ones, so that the new entities **inherit** features and functionality from their prototypes, and perhaps add own features and functionality.



Inheritance 1

Define new entities based on existing ones, so that the new entities **inherit** features and functionality from their prototypes, and perhaps add own features and functionality.

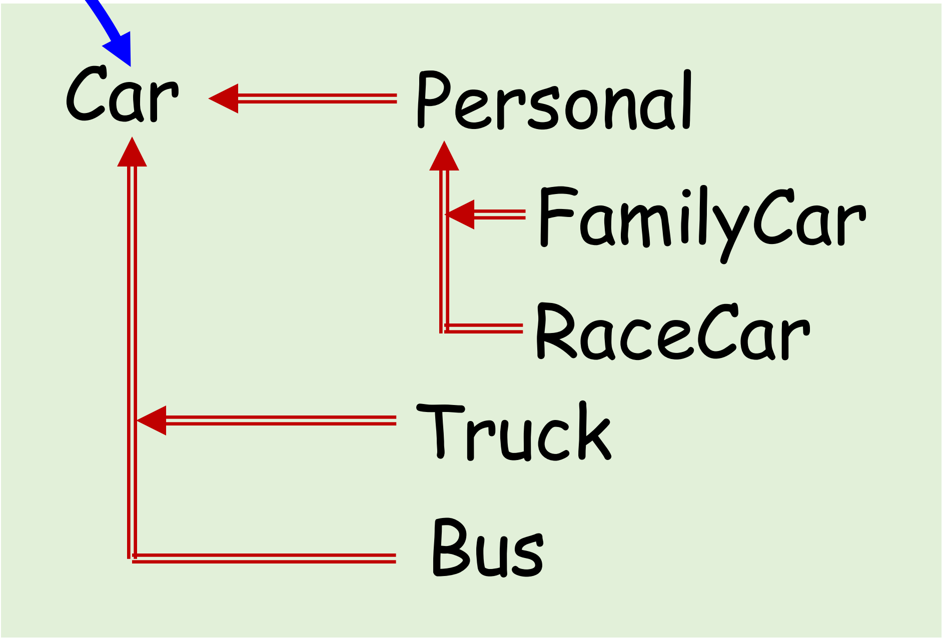




Inheritance 2

"Car" defines features **common** to all kinds of cars, e.g.:

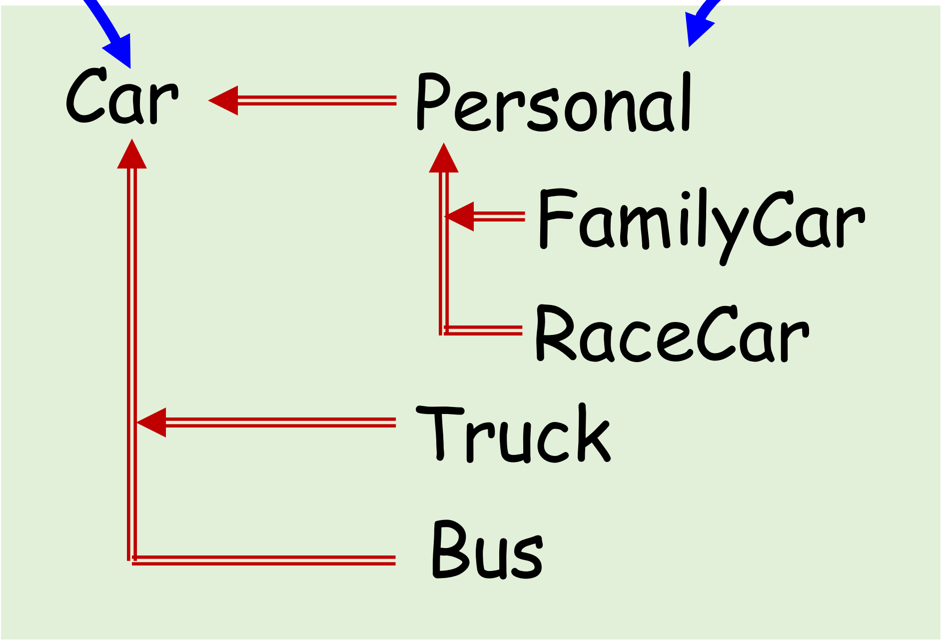
- Max. speed
- Engine
- Capacity
- Acceleration
- Etc.



Inheritance 2

"Car" defines features **common** to all kinds of cars, e.g.:

- Max. speed
- Engine
- Capacity
- Acceleration
- Etc.



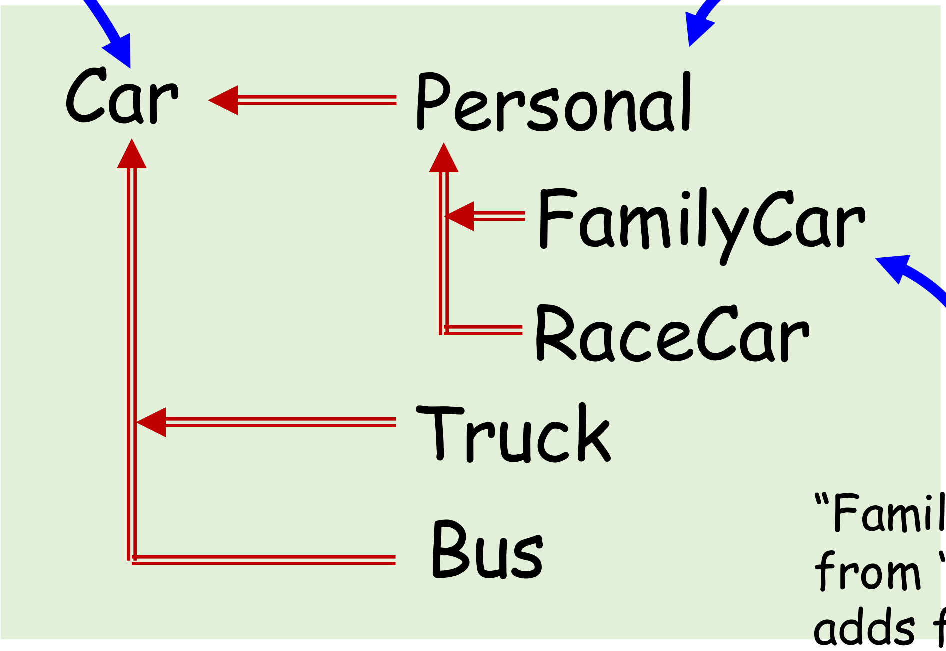
"Personal" **inherits** all features from Car and adds features specific for personal cars, e.g.:

- No. of passengers
- Kind of transmiss.
- Etc.

Inheritance 2

"Car" defines features **common** to all kinds of cars, e.g.:

- Max. speed
- Engine
- Capacity
- Acceleration
- Etc.



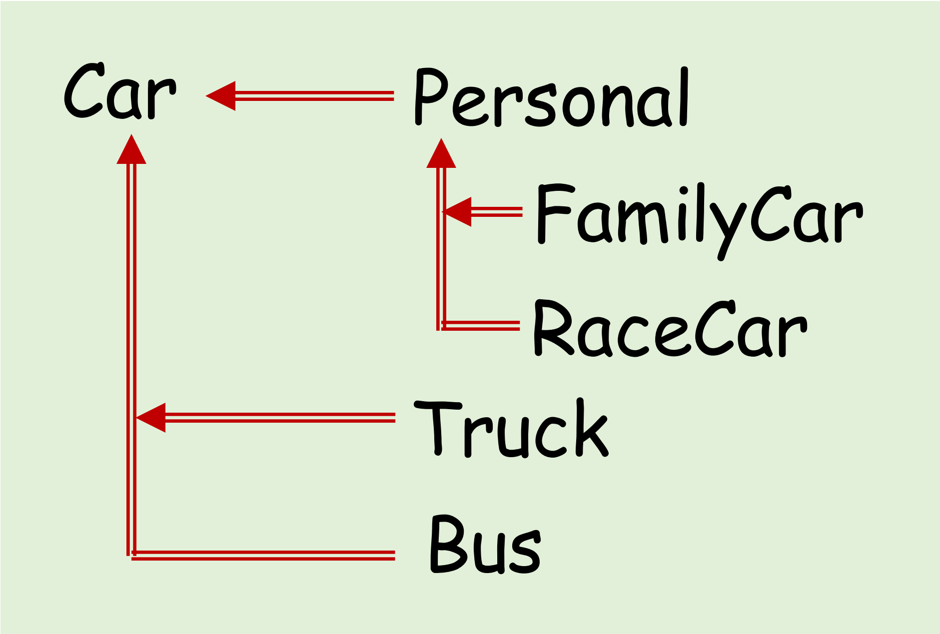
"Personal" **inherits** all features from Car and adds features specific for personal cars, e.g.:

- No. of passengers
- Kind of transmiss.
- Etc.

"FamilyCar" inherits all features from "Car" and "Personal" and adds features specific for family cars, e.g.:

- Seats for children
- Navigator
- Etc.

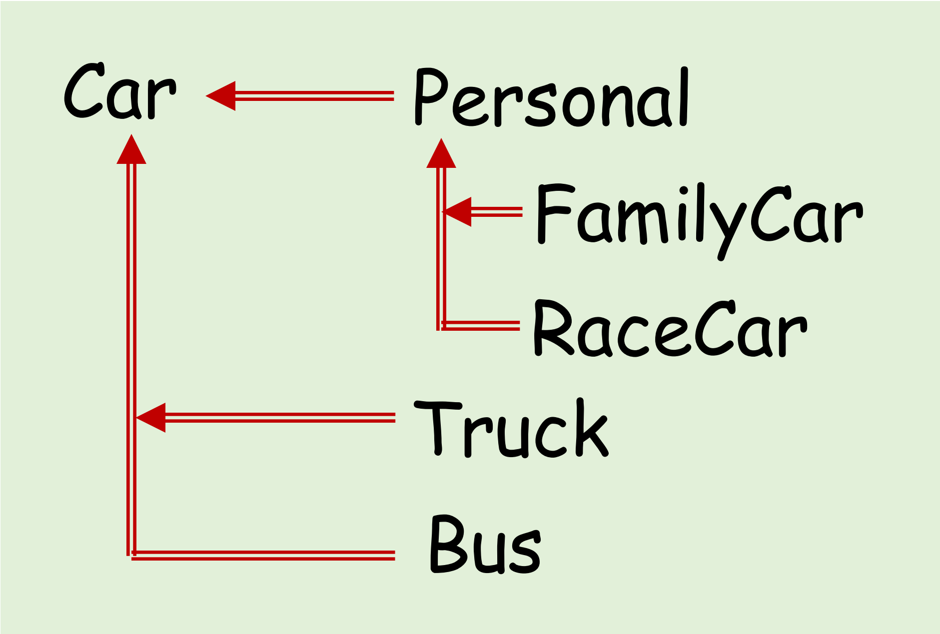
Inheritance 3



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

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

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".

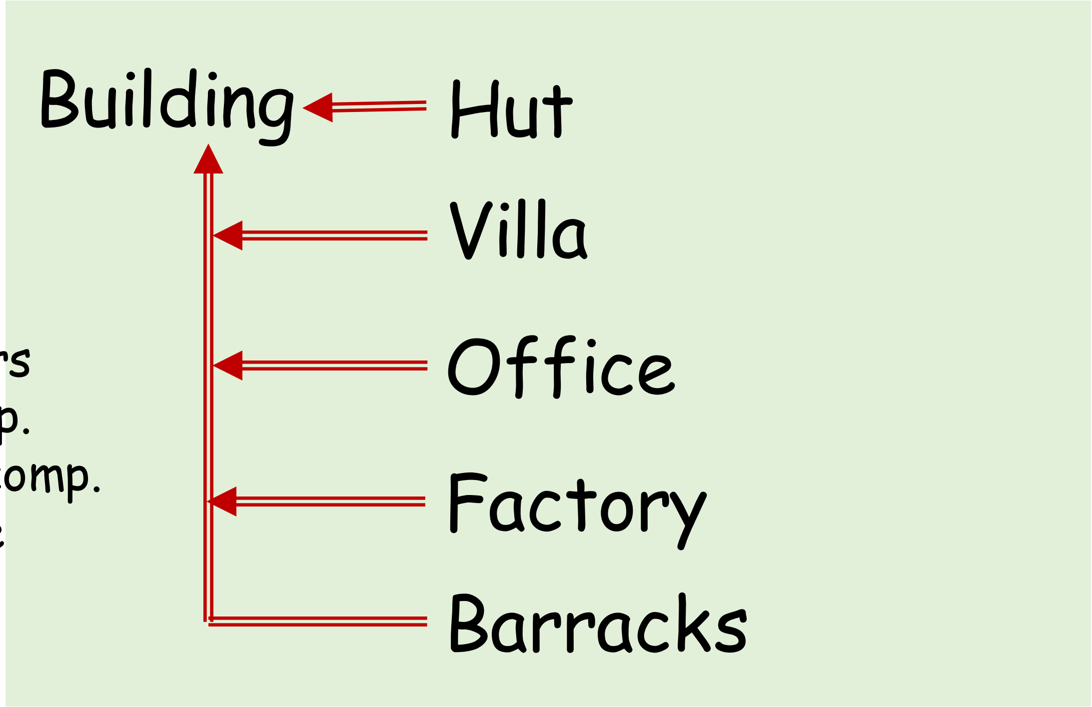
Inheritance: Single & Multiple

- **Single** inheritance: C#, Java, Scala
  - Simple & easy to understand
  - More efficient in implementation
  - Less powerful ("interfaces" help to overcome)

Inheritance: Single & Multiple

- **Single** inheritance: C#, Java, Scala
  - Simple & easy to understand
  - More efficient in implementation
  - Less powerful ("interfaces" help to overcome)
- **Multiple** inheritance: C++, Eiffel ...and Python ☺
  - Harder to understand; causes problems while maintenance
  - A bit less efficient
  - **More common and powerful**

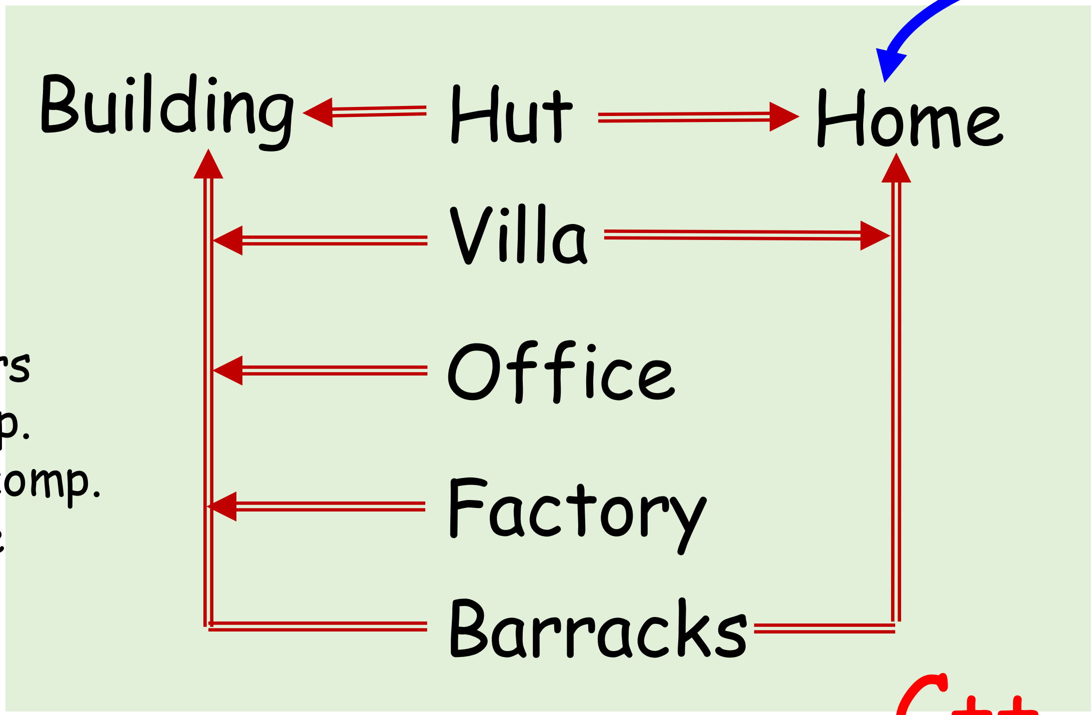
Multiple Inheritance



No of stocks
No of elevators
Architec. comp.
Maintenance comp.
Overall square
No of apts
Etc.

Multiple Inheritance

No of stocks
No of elevators
Architec. comp.
Maintenance comp.
Overall square
No of apts
Etc.



No of rooms
Square meters
Balcony?
Kitchen equipped?
No of bathrooms
Current cost
Etc.

C++, Eiffel

"Villa" **<u>is a</u>** "Building" and **<u>is</u>** "Home" at the same time

Inheritance: The Terminology

Java

```
class B { ... }
class A extends B { ... }
```

Inheritance: The Terminology

```java
class B { ... }
class A extends B { ... }
```

Java

There are several synonyms for inheritance.
When **class** A **extends** **class** B, we can also say that:

- **class** A **inherits** from **class** B
- **class** A **is a subclass of class** B
- class A is a **derived** class for **class** B
- **class** A **is a child of class** B
- **class** A **refines class** B
- class B **is the base class for** A
- **class** B **generalizes class** A
- **class** B **is the parent of class** A
- class B **is a superclass for class** A

Inheritance: The Terminology

```java
class B { ... }
class A extends B { ... }
```

Java

There are several synonyms for inheritance.
When **class** A **extends** **class** B, we can also say that:

class A **inherits** from **class** B
class A is a subclass of class B
class A is a **derived class** for **class** B
class A **is a child of class** B
class A **refines class** B
class B **is the base class for** A
class B **generalizes class** A
class B **is the parent of class** A
class B **is a superclass for class** A

Terminology chosen for Java

Terminology chosen for C++

Single Inheritance 1

The main idea

```
class Car
{
   // Features & functionality
   // typical to all kinds of cars
   int Wheels;
   int Power;
    ...
}
```

```
class Truck extends Car
{
   // Features & functionality
   // typical to all kinds of cars
   // (inherited from the Car class)

   // Features & functionality
   // specific to Trucks
   int cargoWeight;
   ...
}
```

C#, **Java**, Scala

Single Inheritance 2

The "subobject" notion

```
class Base
{
   // Members
   // of class Base
}
```

```
class Derived extends Base
{
   // Members
   // of class Derived
}
```

```
class Other
{
   void f() {
      Derived d = new Derived()
   }
}
```

Single Inheritance 2

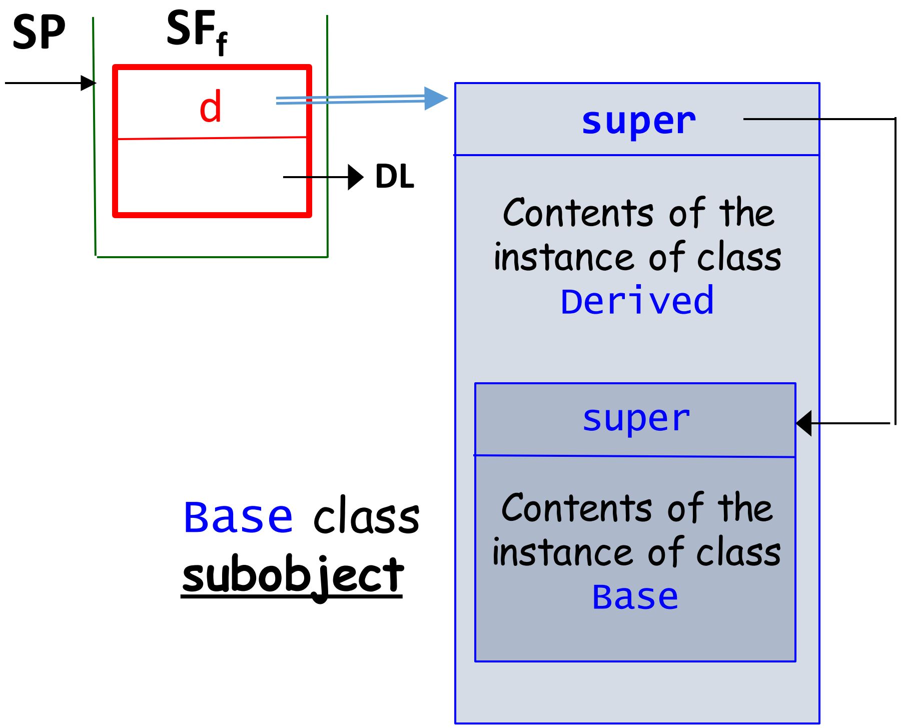
The "subobject" notion

```
class Base
{
   // Members
   // of class Base
}
```

```
class Derived extends Base
{
   // Members
   // of class Derived
}
```

```
class Other
{
   void f() {
     Derived d = new Derived()
   }
}
```

The structure of objects of class Derived:



Single Inheritance 3

The problem with members of the same name

```
class Base
{
   public int m1, m2;
}
```

```
class Derived extends Base
{
   public int m1; // hides Base's m1

   public int f1() { return m1; }

}
```

Normally, attributes in derived classes hide attributes with the same names in derived classes.

Single Inheritance 3

The problem with members of the same name

```
class Base
{
   public int m1, m2;
}
```

```
class Derived extends Base
{
   public int m1; // hides Base's m1

   public int f1() { return m1; }
   public int f2() { return super.m1; }
}
```

Normally, attributes in derived classes hide attributes with the same names in derived classes.

BUT: there is a way to get access to attribute from the base class.

How to access to m1 from the superclass? –
use the keyword **super**

(C# uses the keyword **base** for this)

Single Inheritance 3

The problem with members of the same name

```
class Base
{
   public int m1, m2;
}
```

```
class Derived extends Base
{
   public int m1; // hides Base's m1

   public int f1() { return m1; }
   public int f2() { return super.m1; }
}
```

Normally, attributes in derived classes hide attributes with the same names in derived classes.

BUT: there is a way to get access to attribute from the base class.

The same as

```
public int f1() { return this.m1; }
```

How to access to m1 from the superclass? –
use the keyword **super**
(C# uses the keyword **base** for this)

Single Inheritance 4

The problem with access to private members

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

```
class Derived extends Base
{
   public int f1() { return m1; }
}
```

Error: access to a private attribute

m1 is **private**…

…therefore, m1 is **not accessed** in the derived class

Single Inheritance 4

The problem with access to private members

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

```
class Derived extends Base
{
   public int f1() { return m1; }
}
```

Error: access to a private attribute

`m1` is **private**...

...therefore, `m1` is **not accessed** in the derived class

Possible solution: make `m1` **`public`**.
Is it a good solution?..

Single Inheritance 5

The problem with access to private members

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

```
class Derived extends Base
{
   public int f1() { return m1; }
}
```

OK!

Solution:

<u>protected members</u>

Class members declared as **protected** are accessible (only) **in derived classes**

Single Inheritance 6

The problem with access to private members

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

Version 1

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

Single Inheritance 6

The problem with access to private members

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

```
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.

Single Inheritance 6

The problem with access to private members

```
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.

Single Inheritance 6

The problem with access to private members

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`**.

Access Rules for Class Members

- `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 <u>without a specifier </u>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.

Method Overriding

The case with methods of the same name in base and derived classes

```
class Base
{
    void f(int x) { ... }
}
```

```
class Derived extends Base
{
    void f(int x) { ... }
}
```

For functions with the same signature in base and derived classes neither hiding nor overloading rule applies:

Instead the rule is:

The function in the derived class **overrides** the function with the same signature from the base class

We will see what does it mean soon...

Static & Dynamic Types 1

```
class Shape
{
   ...
};

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

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

Basic OOP rule:

- **Object of the derived type can be converted 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.

Static & Dynamic Types 1

```
class Shape
{
   ...
};

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

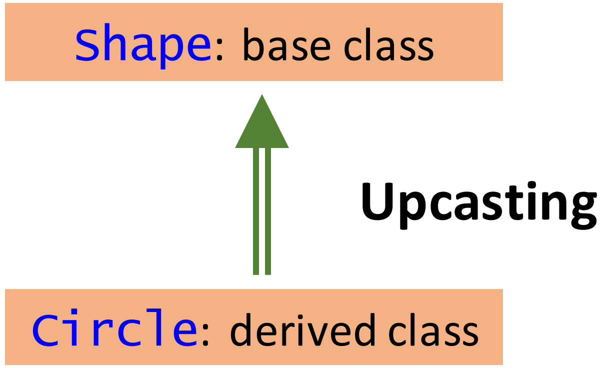
```
Circle circle = new Circle();
...
Shape shape = circle;
```

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`.



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

Static & Dynamic Types 2

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

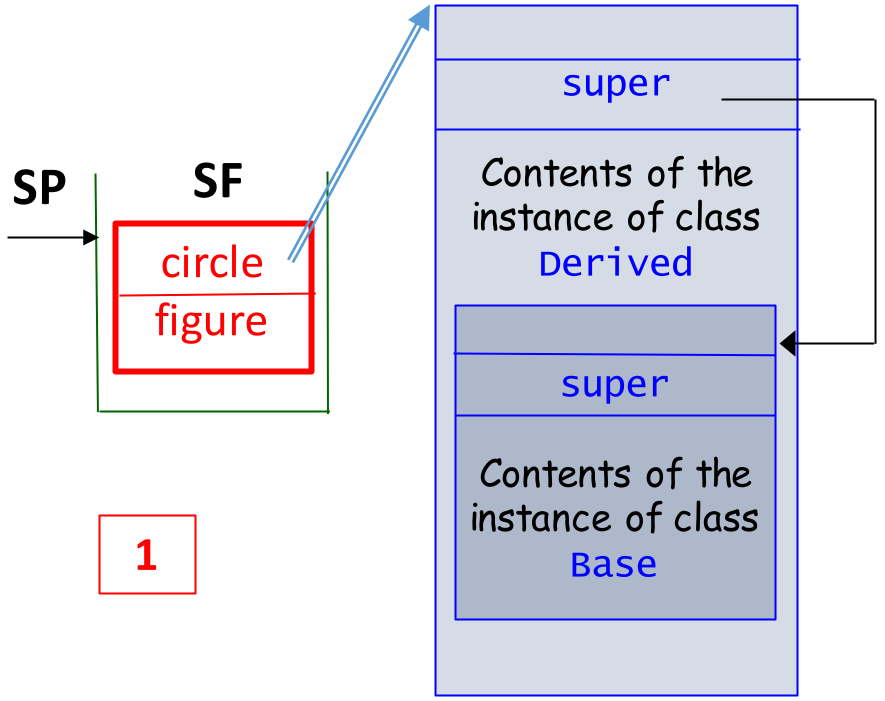
```
Circle circle = new Circle();

...

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`.

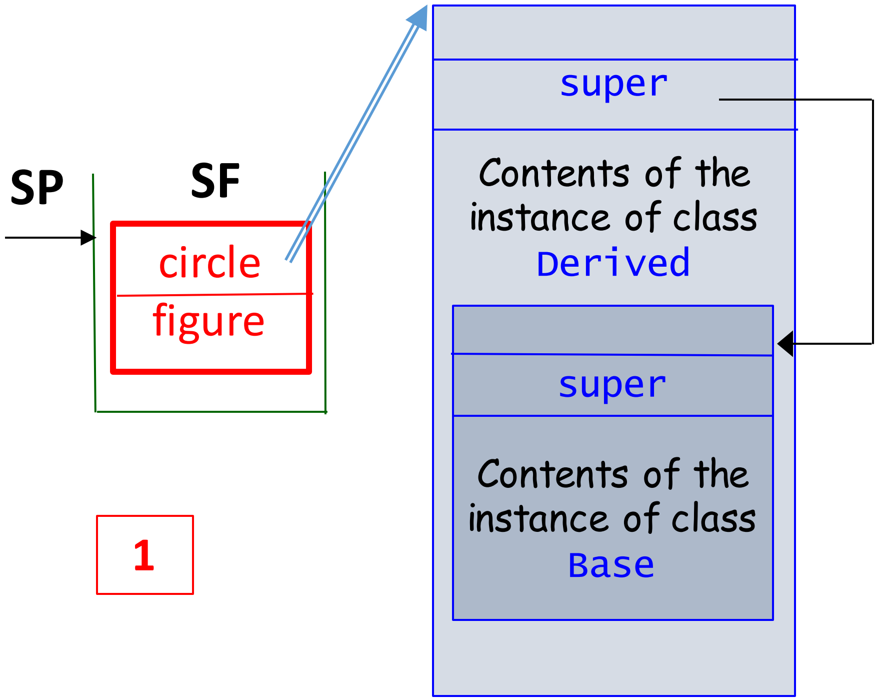
Static & Dynamic Types 3

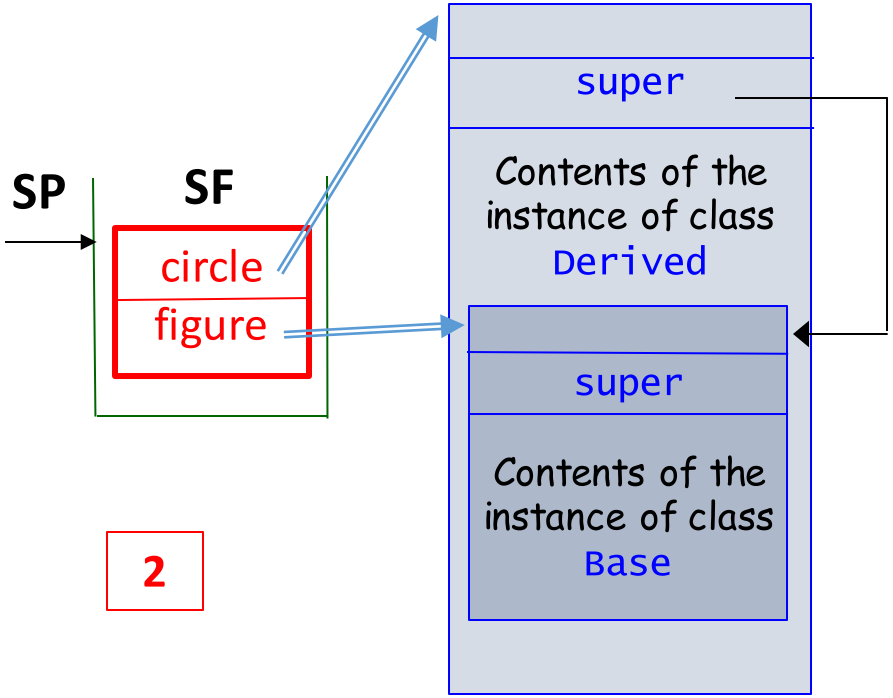
```
(1)    Circle circle = new Circle();
       ...
(2)    Shape figure = circle;
```



Static & Dynamic Types 3

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





"Polymorphism" is from Greek

- πολύς, polys: "many, much"

and

- μορφή, morphē: "form, shape"

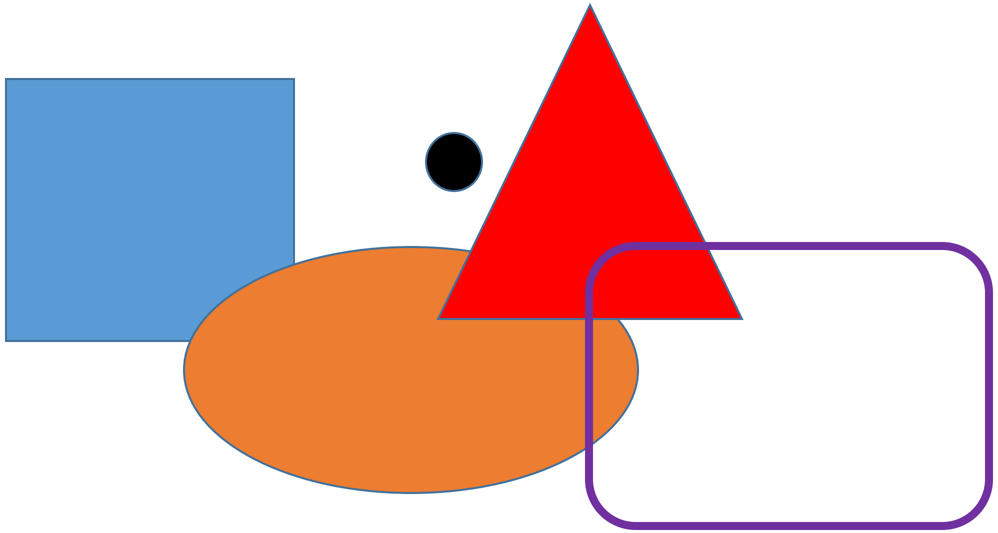
Polymorphism

The third cornerstone of object orientation (*after encapsulation and inheritance*)

Polymorphism 1

Let's start with a simple example:

- Suppose we have a set of various figures on the screen (or on the table) – triangle, square, rectangle, circle etc. We would like to define some operations on those figures: move figures around the table, rotate them, increase their size etc.



What's the conventional ("procedural") solution?

- Represent each figure as a structure and define necessary operations for each structure.

Polymorphism 2

```
class Shape
{
  int code;
  // Shape attributes:
  // Size, coordinates,
  // color, etc.
  public Shape(int c, ...)
  {
     ...
  }
}
```

"Procedural" (not OOP) solution

```
// Operations
// for Triangle:
void moveTriangle(Shape f);
void rotateTriangle(Shape f);
void increaseTriangle(Shape f);
...
```

```
// Operations
// for Cirle:
void moveCircle(Shape f);
void rotateCircle(Shape f);
void increaseCircle(Shape f);
...
```

```
// Operations
// for Rectangle:
void moveRect(Shape f);
void rotateRect(Shape f);
void increaseRect(Shape f);
...
```

Polymorphism 3

How to work with a set of figures?

```
...
Shape[] figures = new Shape[20];
figures[0] = new Shape(1,...); // circle
figures[1] = new Shape(2,...); // triangle
...
```

How to increase sizes
for all figures on the table?

Polymorphism 3

How to work with a set of figures?

```
...
Shape[] figures = new Shape[20];
figures[0] = new Shape(1,...); // circle
figures[1] = new Shape(2,...); // triangle
...
```

How to increase sizes
for all figures on the table?

```
void increaseFigures {
  for ( int i=0; i<20; i++ )
  {
     Shape fig = figures[i];
     switch ( fig.code ) {
       case 1 : increaseCircle(fig); break;
       case 2 : increaseTriangle(fig); break;
       ...
     }
  }
}
```

Polymorphism 3

How to work with a set of figures?

```
...
Shape[] figures = new Shape[20];
figures[0] = new Shape(1,...); // circle
figures[1] = new Shape(2,...); // triangle
...
```

How to increase sizes
for all figures on the table?

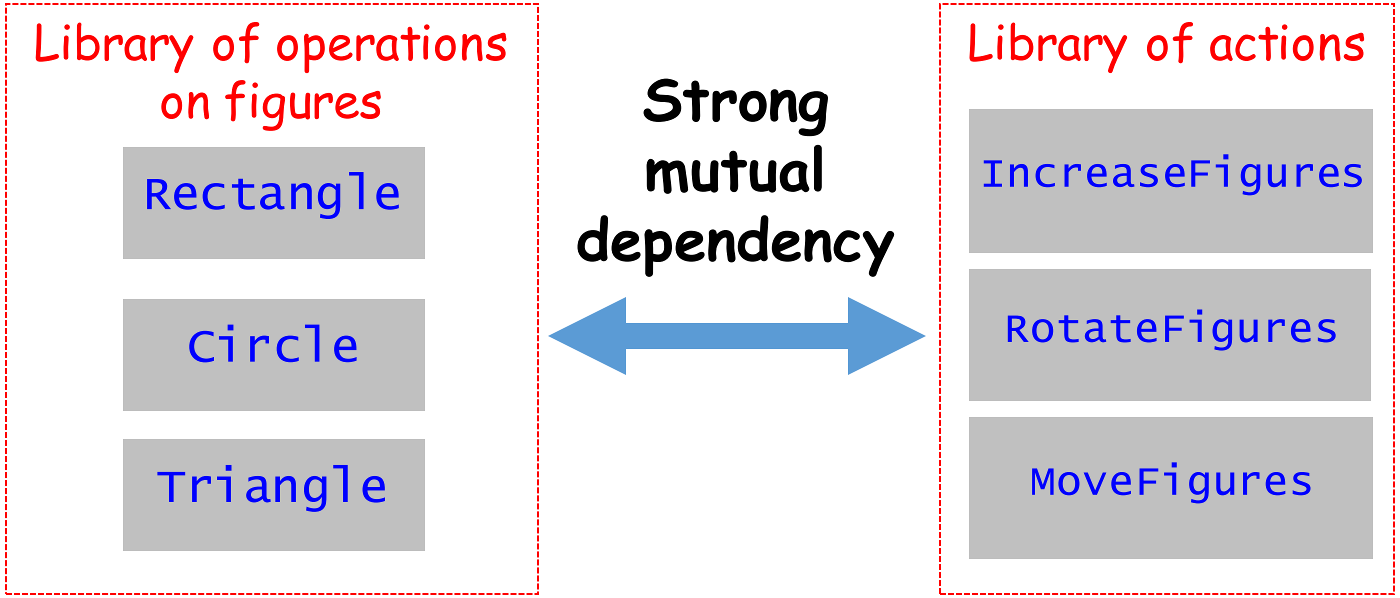
```
void increaseFigures {
  for ( int i=0; i<20; i++ )
  {
     Shape fig = figures[i];
     switch ( fig.code ) {
       case 1 : increaseCircle(fig); break;
       case 2 : increaseTriangle(fig); break;
       ...
     }
  }
}
```

What's the most important disadvantage of such a solution?

Polymorphism 4

Disadvantages:

- Error-prone: a lot of similar code
- Hard to read and maintain
- What to do if we add a new figure?- **Hard to improve**! (The most important)



Polymorphism 5

```
class Shape
{
  // Data common to all shapes
  Coords coords;
  ...  // Size etc.
  // Behavior common to all shapes
  void Move() { }
  void Rotate() { }
  void Draw() { }
  void Increase() { }
  ...
};
```

Object-oriented solution

The main step:

Building class hierarchy...

```
class Rectangle extends Shape
{
  // Data specific to rectangles
  ...
  // Behavior of rectangles
  void Move() { ... }
  void Rotate() { ... }
  void Draw() { ... }
  void Increase() { ... }
  ...
};
```

...Refactoring common actions and attributes

Polymorphism 6

```
class Shape
{...void Increase()...}
```

```
class Circle extends Shape
{...void Increase()...}
```

```
class Rectangle extends Shape
{...void Increase()...}
```

...

```
class Square extends Rectangle
{...void Increase()...}
```

Object-oriented solution

The effect:

The functions in derived classes **don't hide** the ones from the base class, but **override** them.

- **What does it mean?**

Polymorphism 7

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

Small remark:
In Java, all methods are by default **virtual**.

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

```
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 <u>overrides</u> the method with the same signature from the base class

Polymorphism 7

```
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);
  }
}
```

The static type of m is (always) Base

Polymorphism 7

```
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 <u>overrides</u> 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`

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 <u>overrides</u> 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);
  }
}
```

The static type of m is (always) Base

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

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

Polymorphism 8

How to work with a set of figures?

```
...
Shape[] figures = new Shape[20];
figure[0] = new Cicrle();
figure[1] = new Rectangle();
...
```

How to increase sizes
for all figures on the table?

Polymorphism 8

How to work with a set of figures?

```
...
Shape[] figures = new Shape[20];
figure[0] = new Cicrle();
figure[1] = new Rectangle();
...
```

How to increase sizes
for all figures on the table?

```
void IncreaseFigures {
  for ( int i=0; i<20; i++ )
  {
    figures[i].Increase()
  }
}
```

Polymorphism 8

How to work with a set of figures?

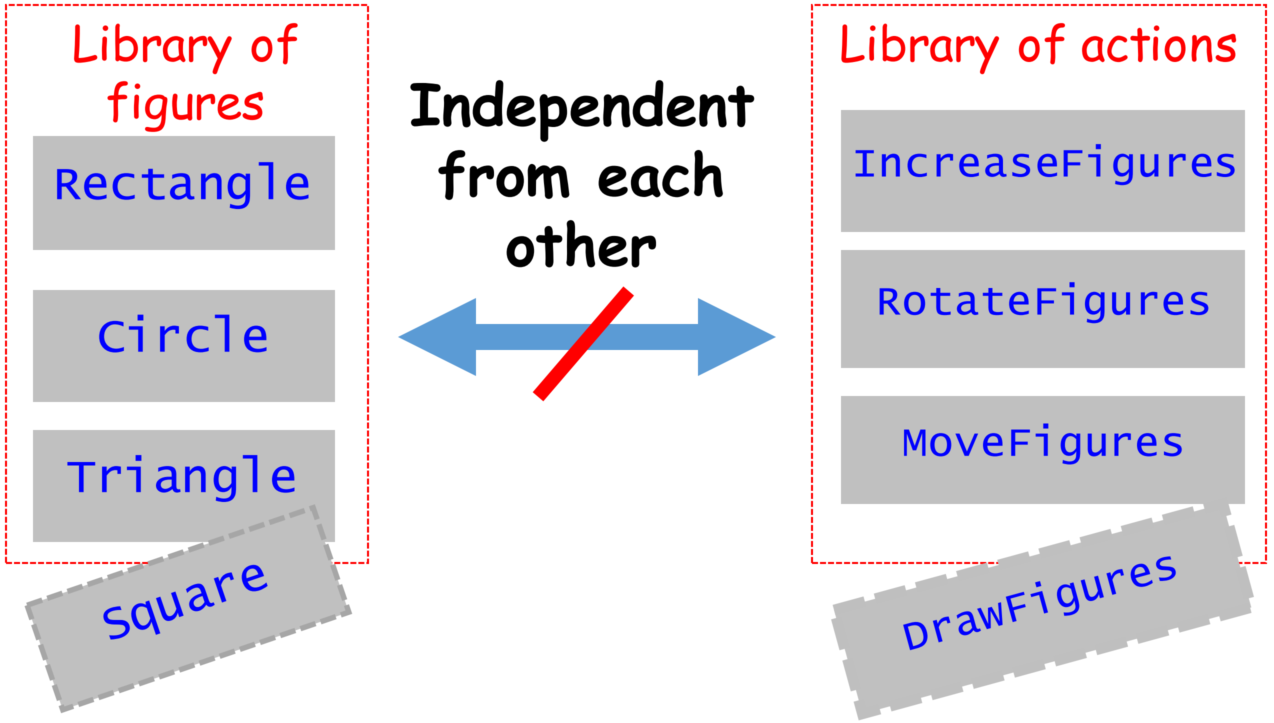
```
...
Shape[] figures = new Shape[20];
figure[0] = new Cicrle();
figure[1] = new Rectangle();
...
```

How to increase sizes
for all figures on the table?

```
void IncreaseFigures {
  for ( int i=0; i<20; i++ )
  {
    figures[i].Increase()
  }
}
```

What's the most important advantage of such a solution?

Polymorphism 9



- We can **add figures**; the action functions remain unmodified and will work with the extended set of figures.
- We **can add actions** without taking into account the concrete set of figures.

Polymorphism:

The ability for derived types to **modify** the behavior of the base type.

Task Description

Write an object-oriented program:

Expression Calculator

It would take an expression from the input, and produce the result of calculations.

Tasks:

1. **Design the class hierarchy representing syntax for expressions.**
2. **Develop functionality for parsing expressions & building AST.**
3. **Implement expression calculation as tree traverser.**

Task Description: Examples

`7`	→	`7`
`1 + 2`	→	`3`
`5 > 9`	→	`0`
`0 & 1`	→	`0`
`(1-7) | 2 + 9`	→	`8`
`1+(26-98)*15+777<28`		

Task Description: Examples

7	→	7
1 + 2	→	3
5 > 9	→	0
0 & 1	→	0
(1-7) | 2 + 9	→	8
1+(26-98)*15+777<28		

Remarks:

Let result of logical and relational operators be integers:

- Op1 & Op2 produces 1 if both operands are not equal 0, and 0 otherwise.
- Op | Op2 produces 0 if both operands are equal to 0, and 1 otherwise.

Treat relations similarly: for example,

- Op1 < Op2, Op1 > Op2, and Op1 = Op2 always produce 0 or 1.

Subtask 1: Class Hierarchy (1)

Expression syntax in EBNF-like notation

```
expression -> logical

logical    -> relation
               { ( "&" | "|" ) relation }

relation   -> term
               [ ( "<" | ">" | "=" ) term ]

term       -> factor
               { ( "+" | "-" ) factor }

factor     -> primary
               { "*" primary }

primary    -> integer | "(" expression ")“

integer    -> Any integer number (literal constant)
```

Legend

{ … } repetition for 0 or more times

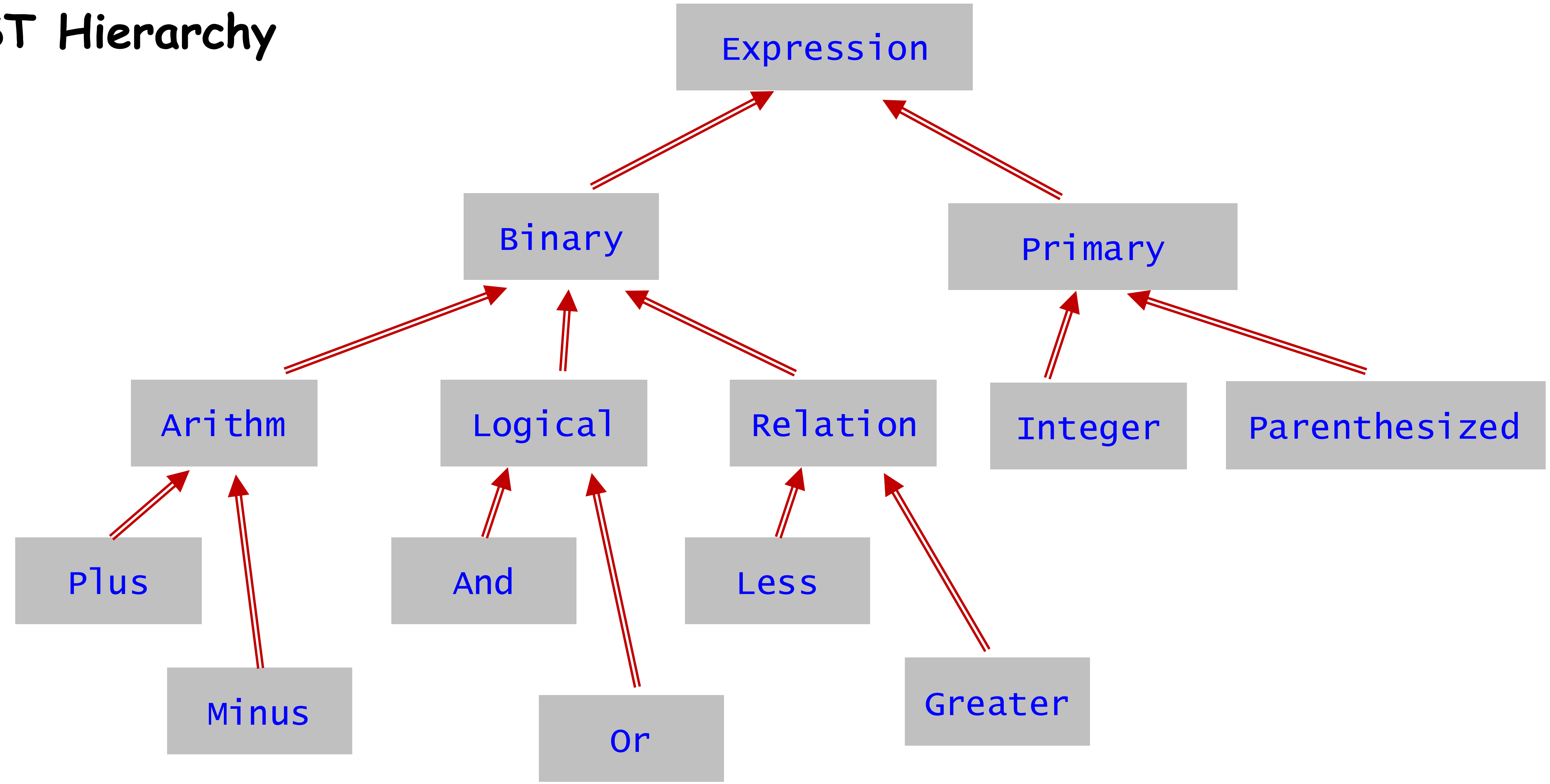
[…] repetition of 0 or 1 times (optional)

| separates alternatives

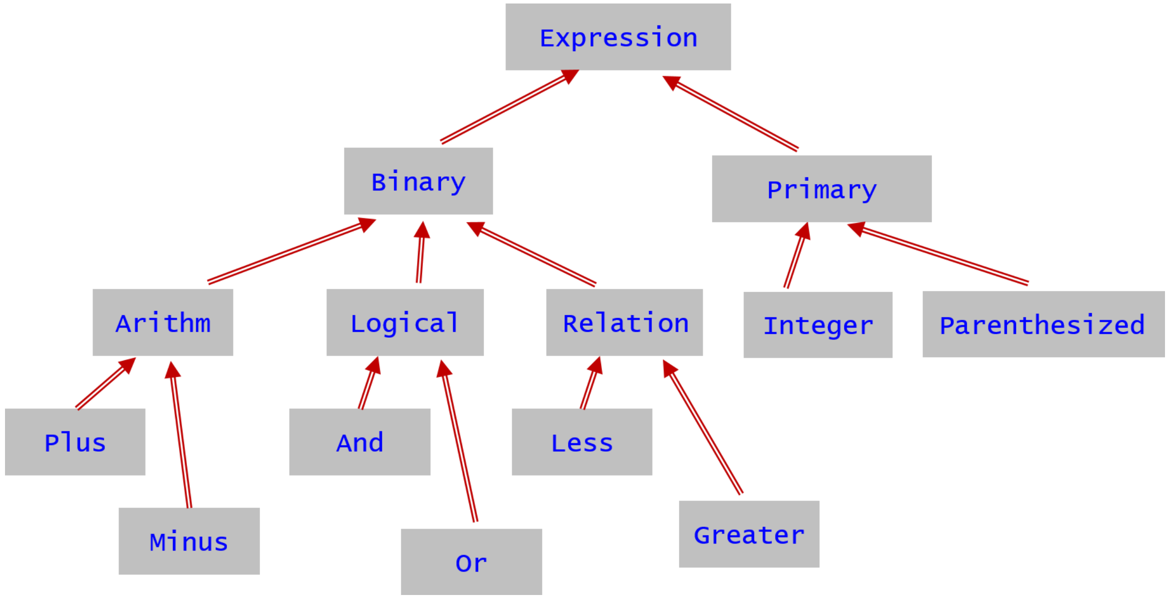
(…) grouping

Subtask 1: Class Hierarchy (2)

AST Hierarchy



Subtask 1: Class Hierarchy (3)




```
class Primary extends Expression
{
}

class Integer extends Primary {
    private long value;
    ...
}

class Parenthesized extends Primary {
    private Expression expression;
    ...
}
```

```
class Expression {
    // Empty for a while
}

class Binary extends Expression
{
    Expression left;
    Expression right;
}

class Logical extends Binary { ... }
    class Or extends Logical { ... }
    class And extends Logical { ... }

class Relation extends Binary { ... }
    class Less extends Relation { ... }
    class Greater extends Relation { ...}
...
```

Subtask 1: Class Hierarchy (4)

Constructors

```
class Binary extends Expression
{
  Expression left;
  Expression right;




}
class Logical extends Binary {




}
class Or extends Logical {




}
...
```

Subtask 1: Class Hierarchy (4)

Constructors

```
class Binary extends Expression
{
  Expression left;
  Expression right;
  public Binary(Expression l, Expression r) {
    left = l; right = r;
  }
}
class Logical extends Binary {
  public Logical(Expression l, Expression r) {
    super(l,r);
  }
}
class Or extends Logical {
  public Or(Expression l, Expression r) {
    super(l,r);
  }
}
...
```

Subtask 1: Class Hierarchy (4)

Constructors

Calls to ctors
of the base class
("superclass")

`new Or(expr1,expr2)`

```
class Binary extends Expression
{
   Expression left;
   Expression right;
   public Binary(Expression l, Expression r) {
     left = l; right = r;
   }
}
class Logical extends Binary {
   public Logical(Expression l, Expression r) {
     super(l,r);
   }
}
class Or extends Logical {
   public Or(Expression l, Expression r) {
     super(l,r);
   }
}
...
```

Subtask 2: Expression Parser (1)

```
class Parser
{
    private string input;
    public Parser(String s) { input = s; }

    public Expression parse() { return parseLogical(); }

    private Expression parseLogical()
    {
        Expression result = parseRelation();
        while ( true ) {
            String op = getNext(); // takes the next token
            Expression right = parseRelation();
            if ( op == "&" )
                result = new And(result,right);
            else if ( op == "|" )
                result = new Or(result,right);
            else
                break;
        }
        return result;
    }
    ...
```

Parser Implementation: Fragment

```
logical   -> relation
                { ( "&" | "|" ) relation }
```

See the previous slide for the idea of how **new** is working

Subtask 2: Expression Parser (2)

Parser Implementation: Fragment

```
class Parser
{
    ...
    private Primary parsePrimary()
    {
        Primary result = null;
        if ( Char.IsDigit(nextChar()) )
            result = parseInteger();
        else if ( nextChar() == '(' ) {
            result = parse();
            nextChar(); // skip ')'
        }
        else
        { ... } // error
        return result;
    }
    ...
}
```

```
primary  -> integer | "(" expression ")"
```

```
private Expression parseInteger()
{
```

```
...
private string input;
private int idx;

public Parser(String s) {
  input = s; idx = 0;
}

char nextChar() {
  return input[idx++];
}
...
```

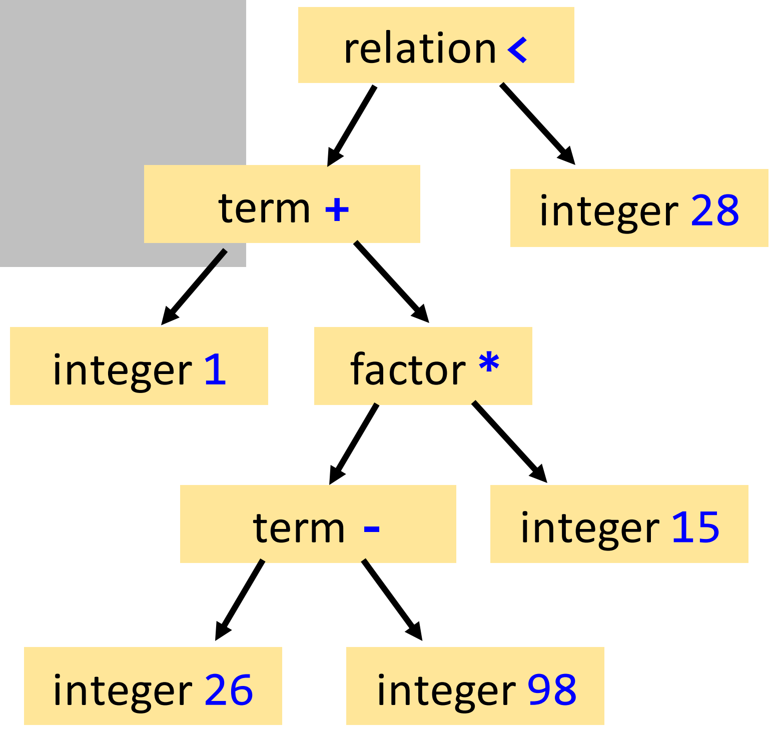
The whole program (1)

```
class Program
{
   public static void main()
   {
      String input = ReadLine(); // 1+(26-98)*15<28
      Parser parser = new Parser(input);
      Expression expressionTree = parser.parse();

   }
}
```

The whole program (1)

```
class Program
{
   public static void main()
   {
       String input = ReadLine(); // 1+(26-98)*15<28
       Parser parser = new Parser(input);
       Expression expressionTree = parser.parse();

   }
}
```

The result of this program
is the expression tree



Subtask 3: Expression Calculator (1)

```
class Expression {

}
class Binary extends Expression {
    Expression left;
    Expression right;






}
...
class Plus extends Term {




}
```

What's the common action for all binary operators?

```
...
class And extends Logical {




}
class Less extends Relational {




}
```

Subtask 3: Expression Calculator (1)

```
class Expression {

}
class Binary extends Expression {
    Expression left;
    Expression right;
    protected tmp1, tmp2;
    protected void calcOperands() {
      tmp1 = left.calculate();
      tmp2 = right.calculate();
    }
}
...
class Plus extends Term {




}
```

What's the common action for all binary operators?
- To calculate values of operands!

What's the specific action for each particular operator?

```
...
class And extends Logical {




}
class Less extends Relational {




}
```

Subtask 3: Expression Calculator (1)

```
class Expression {
    long calculate() { }
}
class Binary extends Expression {
    Expression left;
    Expression right;
    protected tmp1, tmp2;
    protected void calcOperands() {
      tmp1 = left.calculate();
      tmp2 = right.calculate();
    }
}
...
class Plus extends Term {
  long calculate() {
      calcOperands();
      return tmp1+tmp2;
  }
}
```

What's the common action for all binary operators?
- To calculate values of operands!

```
...
class And extends Logical {
   public long calculate() {
      calcOperands();
      return tmp1==0 ? 0 : tmp1 & tmp2;
   }
}
class Less extends Relational {
  public long calculate() {
     calcOperands();
     return (long)(tmp1 < tmp2);
  }
}
```

What's the specific action for each particular operator?
- To execute the action specific to its semantics!

Subtask 3: Expression Calculator (2)

```
...

class Primary extends Expression
{
}

class Integer extends Primary
{
  private long value;
  public long calculate() { return value; }
}

class Parenthesized extends Primary
{
  private Expression expression;
  public long calculate() {
    return expression.calculate();
  }
}
```

The whole program (2)

```
class Program
{
    public static void main()
    {
        String input = ReadLine(); // 1+(26-98)*15<28
        Parser parser = new Parser(input);
        Expression expressionTree = parser.parse();
        long result = expressionTree.calculate();
    }
}
```

The result of this program is <u>the value</u> of the expression tree