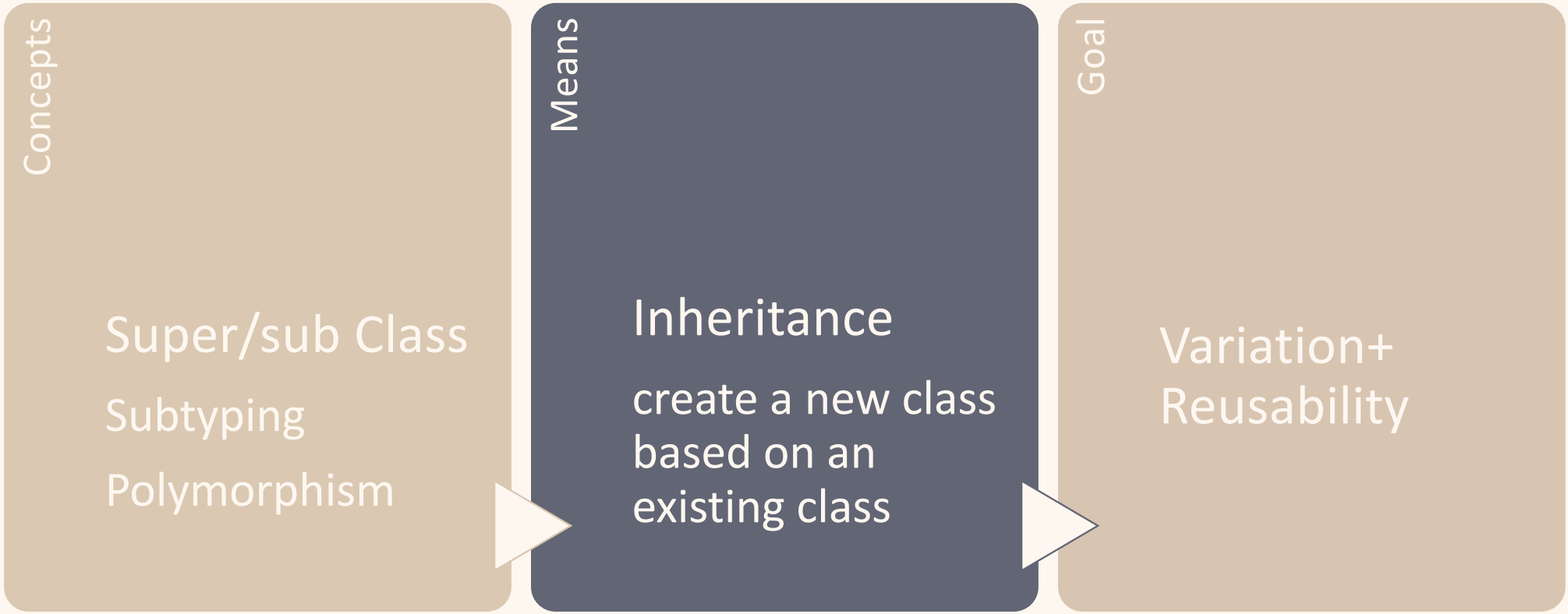
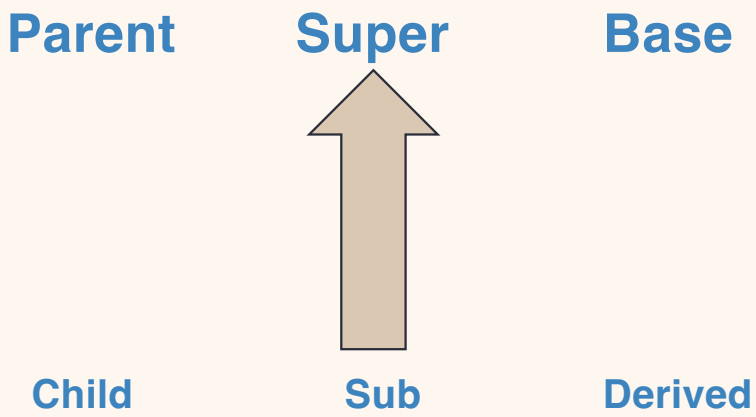
Goal, concepts, means



Inheritance as a solution for code duplication

The solution that inheritance offers is to place the duplicate code in a separate class that is shared in a special way by the old classes.

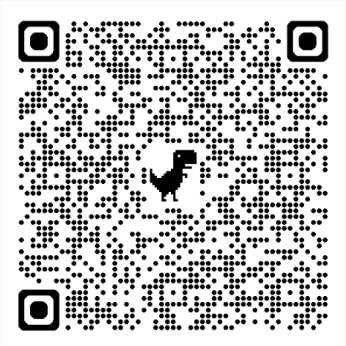
In object orientation, this new class is described as a 'superclass' and the old classes will become 'subclasses'.



Inheritance (Vererbung)

A) Eignen Sie sich ein grundlegendes Verständnis des Konzepts “Vererbung in Java” mit den folgenden Ressourcen an :

- https://www.linkedin.com/learning/java-grundkurs-2-objektorientierte-programmierung-fehlerbehandlung-stream-api/vererbung-und-polymorphie?autoplay=true&resume=false&u=82533698
- https://freiheit.f4.htw-berlin.de/prog1/vererbung/



B) Erstellen Sie eine Liste der neuen Keywords mit einer Erklärung in eigenen Worten.

What is Inheritance?

Inheritance is a fundamental concept of Object-Oriented Programming (OOP).

- Allows one class to inherit properties and methods from another.
- Promotes code reusability and hierarchical classification.

Example: Parent-Child relationships between classes.

Keyword: extends

The keyword 'extends' is used to state that a child class will inherit from another class (the parent).

The child class (e.g. SpecialRoom) inherits from the parent class (Room).

```
class SpecialRoom extends Room {
        // Child-specific properties and methods
}
```

Keyword: super

The keyword 'super' is used in an inheritance hierarchy to refer to the parent (superclass). It allows the child constructor to invoke the parent's constructor and initialize inherited attributes.

Appending parentheses after 'super' calls the parent's constructor.

You can use 'super' to access public (and protected) methods and variables of the parent class.

Access Modifiers and Inheritance

Access modifiers control the visibility of class members.

- private: Not accessible by child classes.
- protected: Accessible within the same package or subclasses.
- public: Accessible from anywhere.

Superclass constructor call

Subclass constructors must always contain a 'super' call.

If none is written, the compiler inserts one (without parameters)

- only compiles if the superclass has a constructor without parameters

Must be the first statement in the subclass constructor.

Method Overriding

Method overriding allows a child class to provide a specific implementation of a method already defined in its parent class.

Rules:

- The method in the child class must have the same name, return type, and parameters.
- The overriding method cannot have a lower access level.

Method Overriding

```
class Animal {
    public void sound() {
        System.out.println("Animals make sounds");
    }
}

class Dog extends Animal {
    public void sound() {
        System.out.println("Dogs bark");
    }
}
```

```
class Cat extends Animal {
    public void sound() {
        super.sound(); //optional, can appear anywhere in method
        System.out.println("Cats meow");
    }
}
```

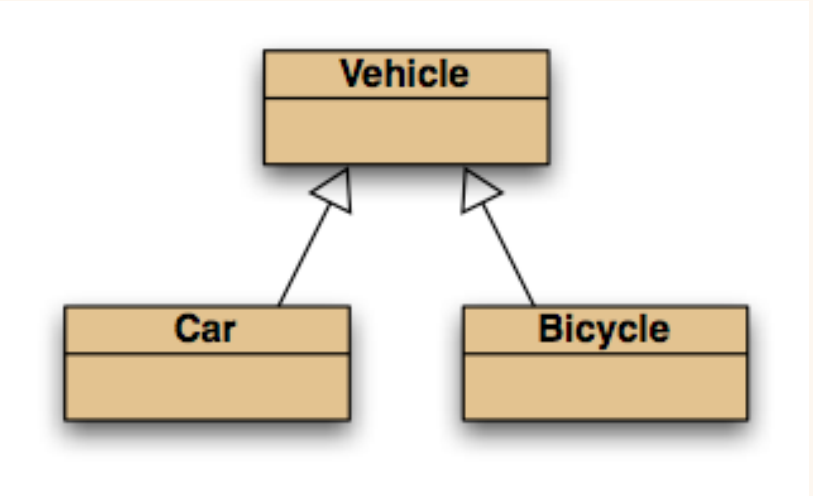
Subclasses and subtyping

Classes define types.

Subclasses define *subtypes*.

Objects of subclasses can be used where objects of supertypes are required. (This is called substitution .)

Subtyping and assignment



subclass objects may be assigned to superclass variables

```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```

Polymorphism: same same but different

Poly = Many , Morph = Forms , (Vielgestaltig)

Do the same in different ways

Polymorphic variables

Object variables in Java are polymorphic.

(They can hold objects of more than one type.)

They can hold objects of the declared type, or of subtypes of the declared type.

Casting

We can assign subtype to supertype …

… but we cannot assign supertype to subtype!

```
Vehicle v;
Car c = new Car();
v = c;   // correct
c = v;   // compile-time error!
```

Casting fixes this:

```
c = (Car) v;
```

(but only ok if the vehicle really is a **`Car`**!)

Casting

An object type in parentheses.

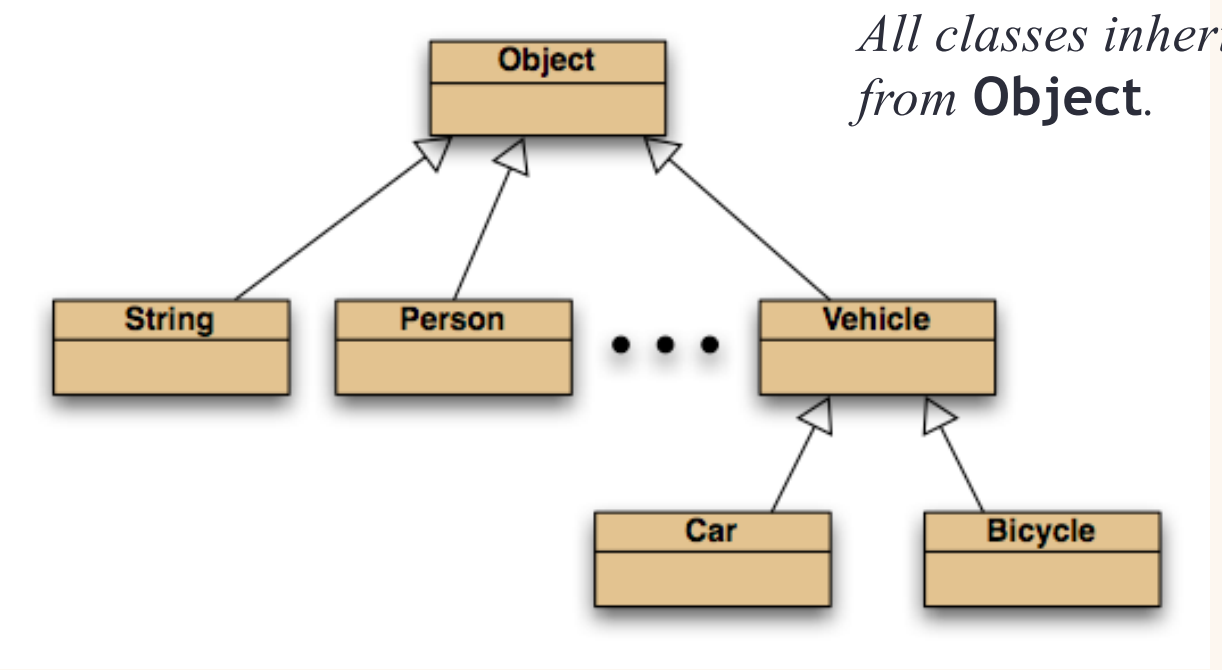
The object is not changed in any way.

A runtime check is made to ensure the object really is of that type:

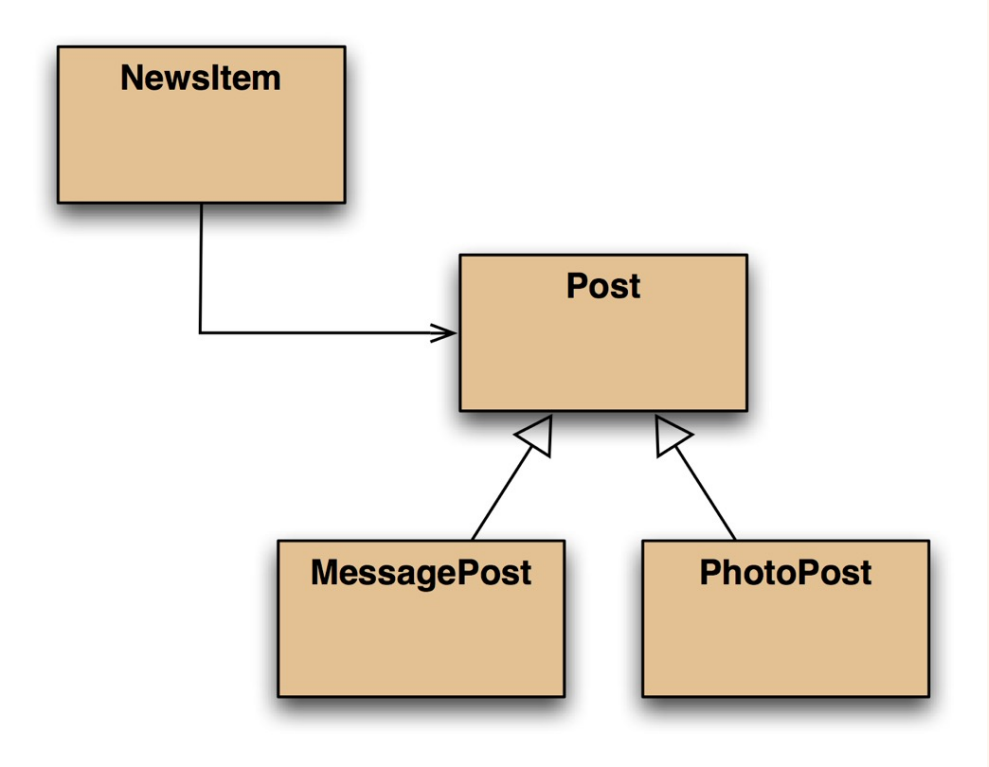
- **`ClassCastException`** if it isn't!

Use it sparingly.

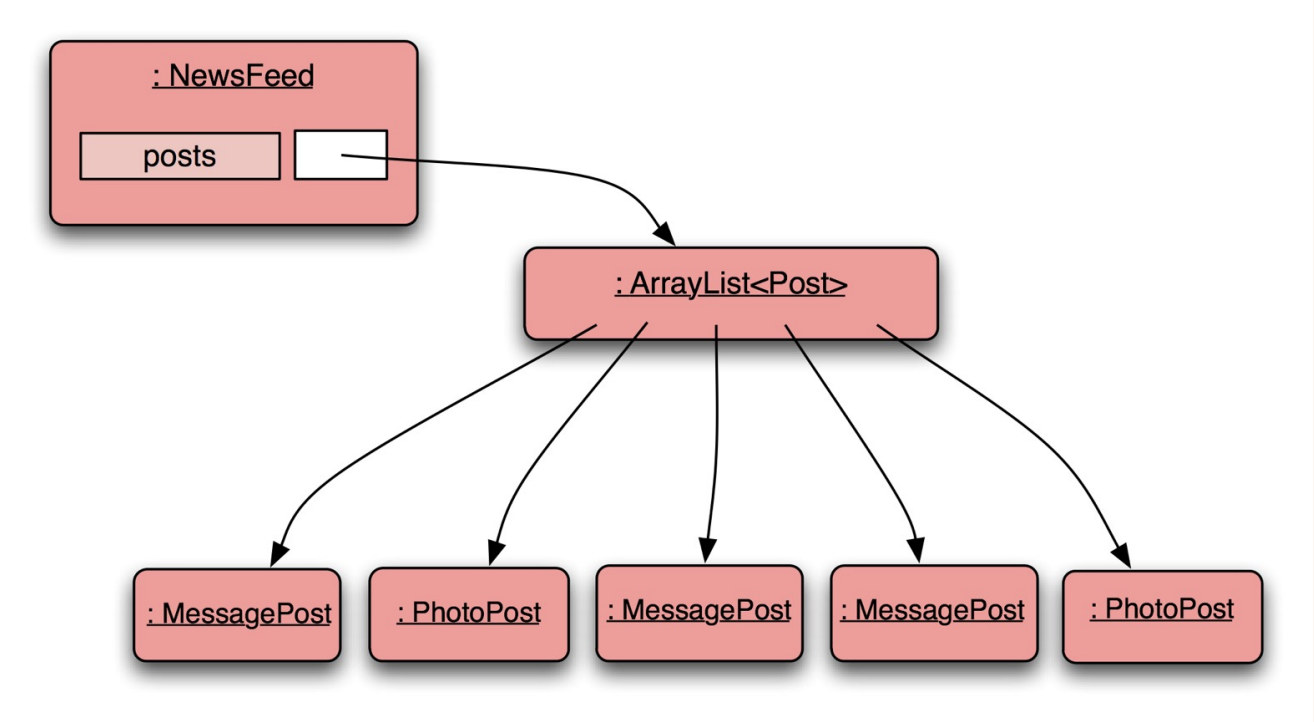
The Object class



Class diagram



Object diagram



Polymorphic collections

All collections are polymorphic.

The elements could simply be of type **`Object`**.

```
public void add(Object element)
```

```
public Object get(int index)
```

Usually avoided by using a type parameter with the collection.

Polymorphic collections

A type parameter limits the degree of polymorphism: **`ArrayList<Post>`**

Collection methods are then typed.

Without a type parameter, **`ArrayList<Object>`** is implied.

Likely to get an *"unchecked or unsafe operations"* warning.

More likely to have to use casts.

Review

Inheritance allows the definition of classes as extensions of other classes.

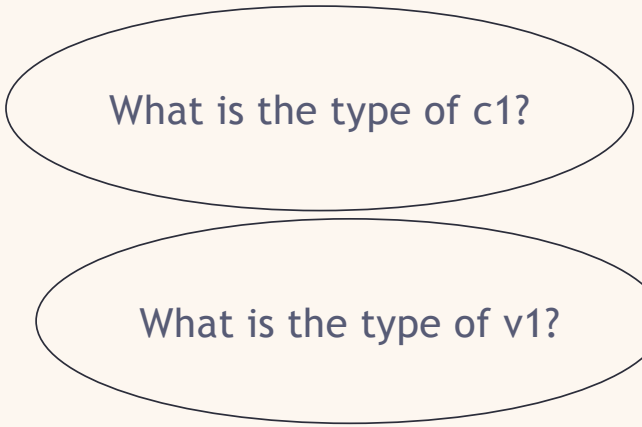
Inheritance

- avoids code duplication
- allows code reuse
- simplifies the code
- simplifies maintenance and extending

Variables can hold subtype objects.

Subtypes can be used wherever supertype objects are expected (substitution).

Static and dynamic type



The **declared type** of a variable is its **static type**.

The type of the object a variable refers to is its dynamic type.

The compiler's job is to check for static-type violations.

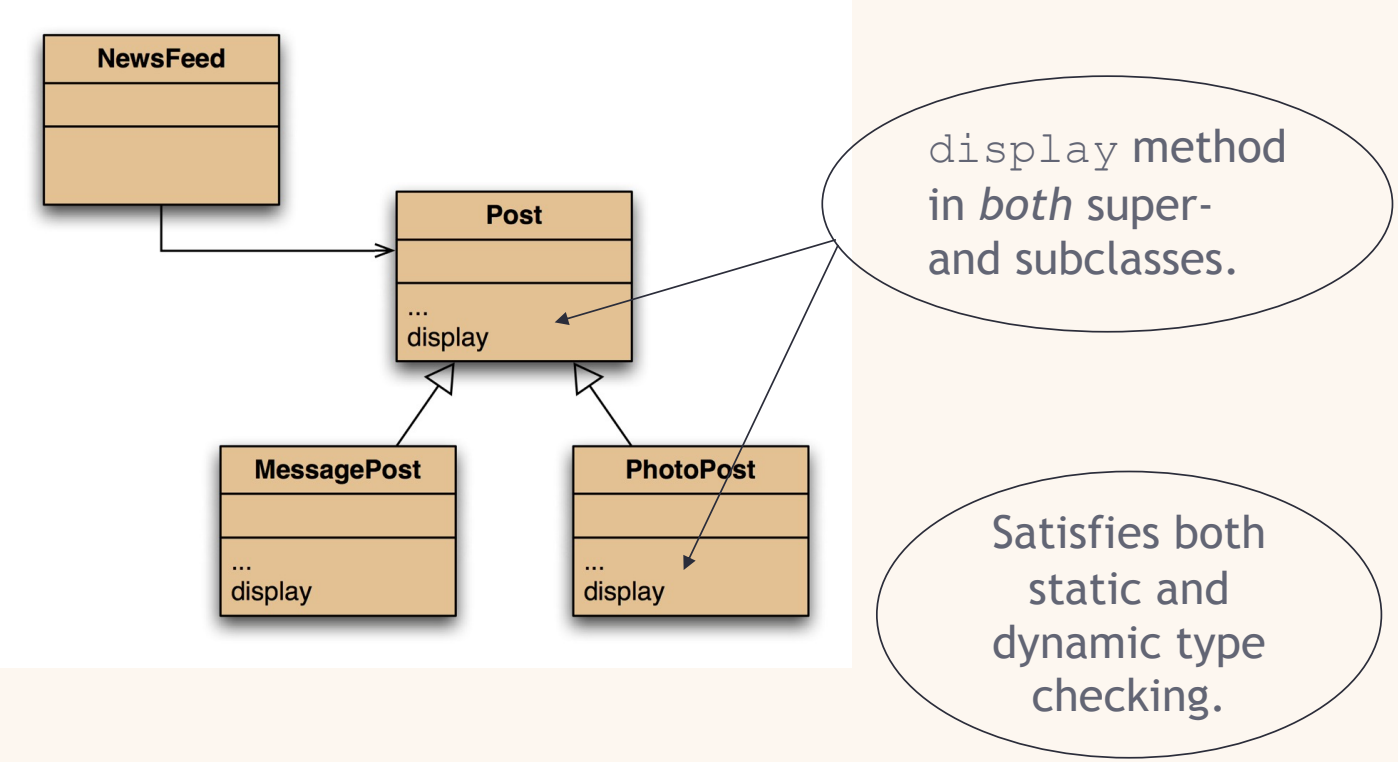
Method polymorphism

A polymorphic variable can store objects of varying types.

Method calls are polymorphic.

- The actual method called depends on the dynamic object type.

Overriding



Overriding

Superclass and subclass define methods with the same signature.

Each has access to the fields of its class.

Superclass satisfies static type check.

Subclass method is called at runtime – it *overrides* the superclass version.

What becomes of the superclass version?

Method Overriding

Method overriding allows a child class to provide a specific implementation of a method already defined in its parent class.

Rules:

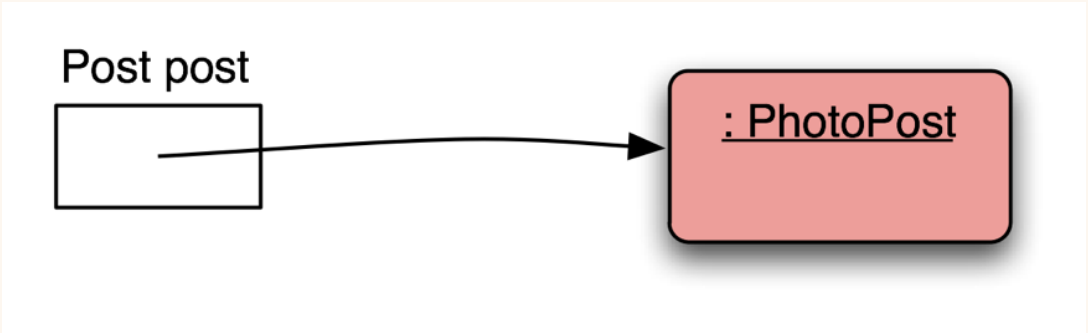
- The method in the child class must have the same name, return type, and parameters.
- The overriding method cannot have a lower access level.

Method Overriding

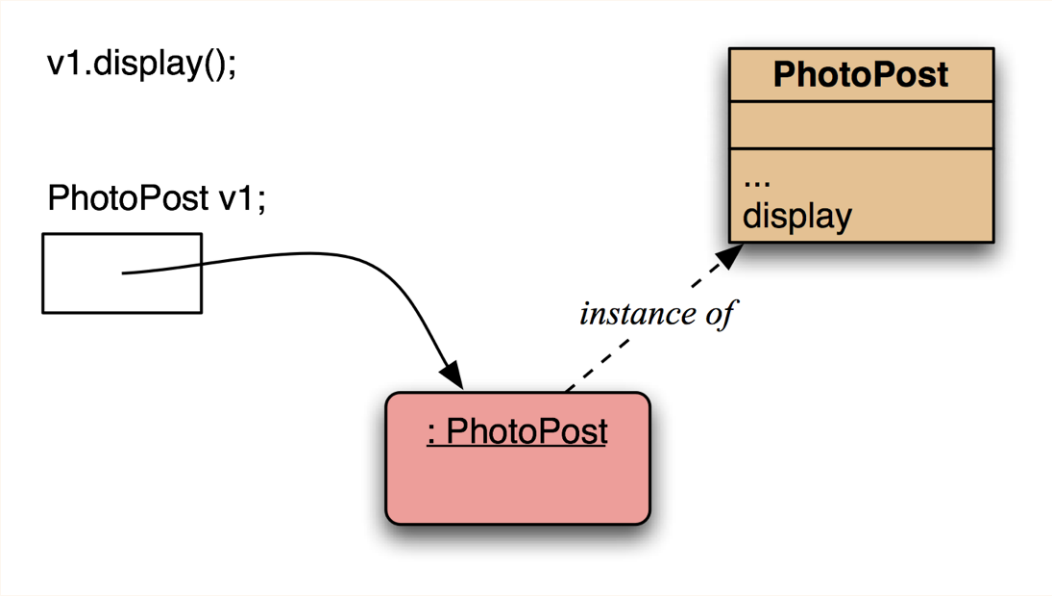
```
class Animal {
    public void sound() {
        System.out.println("Animals make sounds");
    }
}

class Dog extends Animal {
    public void sound() {
        System.out.println("Dogs bark");
    }
}
```

```
class Cat extends Animal {
    public void sound() {
        super.sound(); //optional, can appear anywhere in method
        System.out.println("Cats meow");
    }
}
```

Distinct static and dynamic types

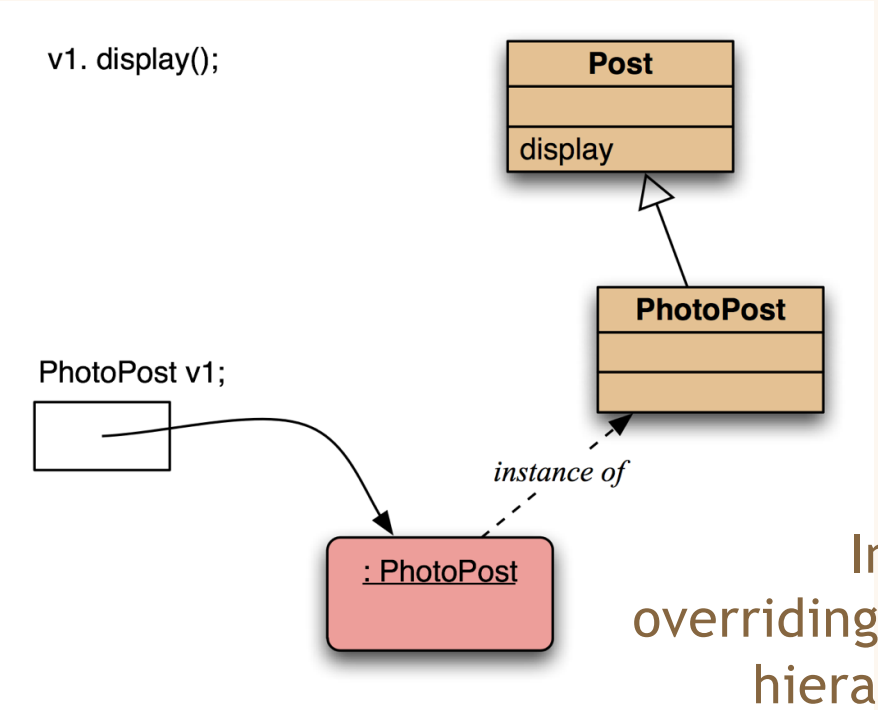


Method lookup



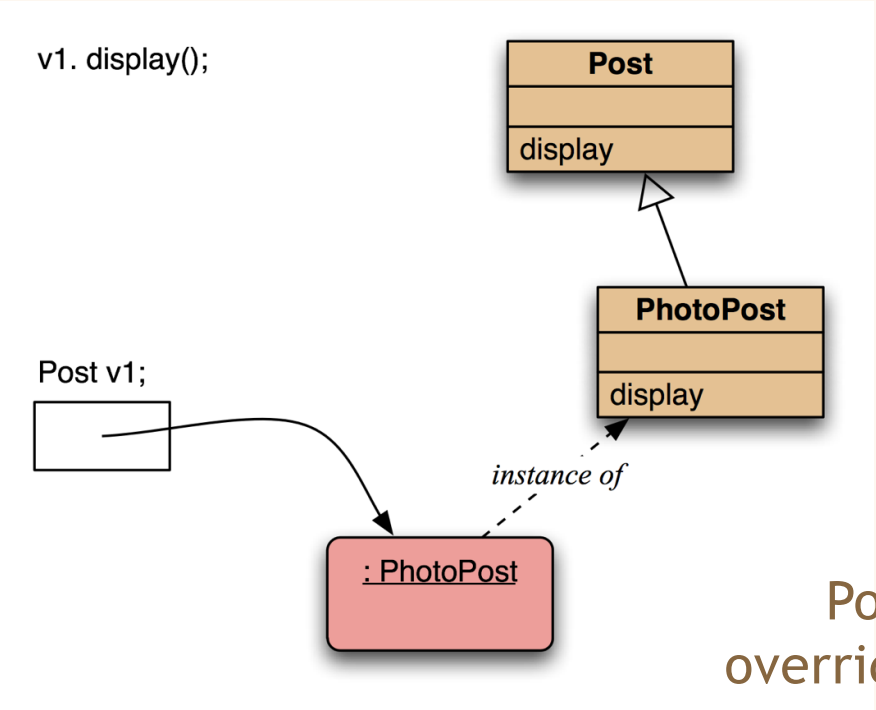
No inheritance or polymorphism.
The obvious method is selected.

Method lookup



Inheritance but no overriding. The inheritance hierarchy is ascended, searching for a match.

Method lookup



Polymorphism and overriding. The ‘first’ version found is used.

Method Overriding

Given the class definitions of **Car** and **Beemer**, what do the following method calls print on the standard output?___

Was wird bei den Klassendefinitionen von **Car** und **Beemer** bei den folgenden Methodenaufrufen auf der Standardausgabe ausgegeben?

```
public class Car
{
  public String noise(){
    return "Vroom, vroom!";
  }
  public void run(){
    while(true)
      System.out.println(this.soundOff());
  }
  public String soundOff(){
    return this.noise();
  }
}
```

```
public class Beemer extends Car
{
  private String startup = "Brumm";
  public String noise(){
    return "Zooooooom!";
  }
  public void run(){
    while (true){
      System.out.println( this.start() + this.soundOff());
    }
  }
  public String start(){
     return this.startup+this.startup;
  }

}
```

The `Object` class's methods

Methods in **`Object`** are inherited by all classes.

Any of these may be overridden.

The **`toString`** method is commonly overridden:

- **`public String toString()`**
- Returns a string representation of the object.

Method lookup summary

The variable is accessed.

The object stored in the variable is found.

The class of the object is found.

The class is searched for a method match.

If no match, the superclass is searched.

This is repeated until a match is found, or the class hierarchy is exhausted.

Overriding methods take precedence – they override inherited copies.

Super call in methods

Overridden methods are hidden ...

... but we often still want to be able to call them.

An overridden method *can* be called from the method that overrides it.

- **`super.method(...)`**
- Compare with the use of **`super`** in constructors.

Keyword: instanceof

https://www.javatpoint.com/downcasting-with-instanceof-operator

Purpose: The instanceof operator checks if an object is an instance of a specific class or its subclass. Enables safe downcasting to specific types only when the type matches.

Return: It returns true if the object belongs to the specified type or can be cast to that type.

Instance of

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal

 public static void main(String args[]){
 Dog1 d=new Dog1();
 System.out.println(d instanceof Animal);//true
 }
}
```

```
class Dog2{
 public static void main(String args[]){
  Dog2 d=null;
  System.out.println(d instanceof Dog2);//false
 }
}
```

Downcasting without instanceof

```
Dog d=new Animal();//Compilation error
```

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

```
Dog d=(Dog)new Animal();
//Compiles successfully but ClassCastException is thrown at runtime
```

Downcasting safely

```
class Animal { }

class Dog3 extends Animal {
 static void method(Animal a) {
  if(a instanceof Dog3){
    Dog3 d=(Dog3)a;//downcasting
    System.out.println("ok downcasting performed");
  }
 }

 public static void main (String [] args) {
  Animal a=new Dog3();
  Dog3.method(a);
 }

}
```

The **`instanceof`** operator

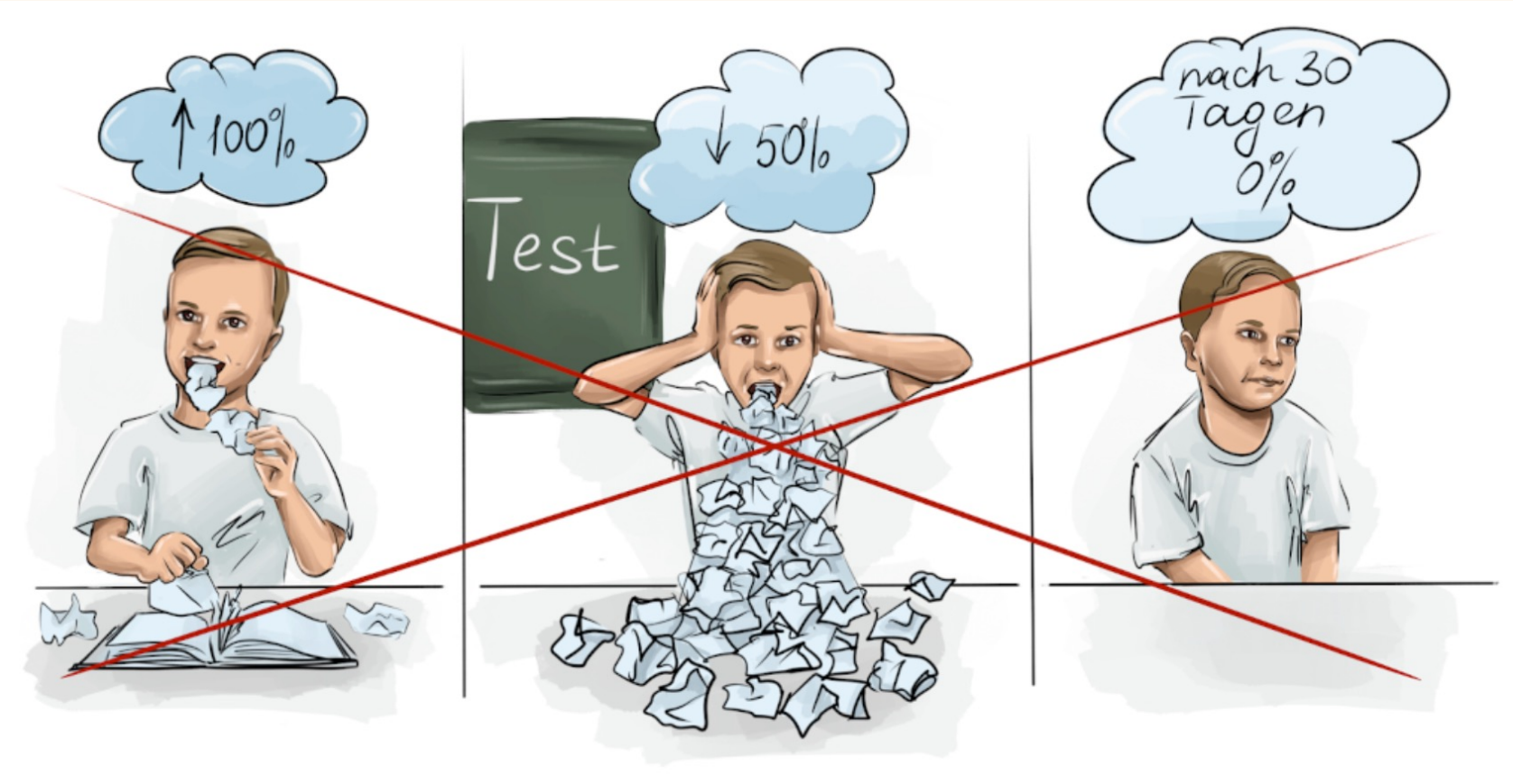
Used to determine the dynamic type.

Identifies 'lost' type information.

Usually precedes assignment with a cast to the dynamic type:

```
if(post instanceof MessagePost) {
    MessagePost msg =
        (MessagePost) post;
    … access MessagePost methods via msg …
}
```

Probeklausur != Tatsächliche Klausur



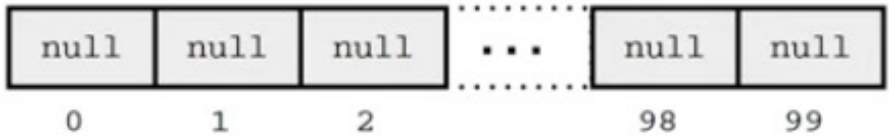
https://wissen.alex-fischer-duesseldorf.de/fundamental-booklet-high-tech-lernen-lehren/bulimie-lernen-vs-verstehen

Beispiele

```
int[] a; // nur Deklaration, kein Speicherplatz reserviert
byte[] b = new byte[12]; // reserviert Speicherplatz für 12 Elemente
String[] s = { "Maier", "Maier", "Maier", "Maier" }; // reserviert Speicherplatz für 4 Elemente und initialisiert diese
```

- **Beispiel:**
  - `double[] a = new double[8];`

  - `String[] b = new String[100];`



  - `int[] p = { 2, 3, 5, 7, 11 };`

Arrays

```
import java.util.Random;
    public void printRandomArray(int num) {
        Random rand = new Random(); // Zufallszahl-Generator
        int[] zahlen = new int[num]; // Array vom Typ int
        // Array mit Zufallszahlen zwischen 0 und 99 füllen
        for (int i = 0; i < zahlen.length; i++) {
            zahlen[i] = rand.nextInt(100); // Zahl von 0 bis 99
        }
        // Ausgabe des Arrays
        System.out.println(”random numbers in array:");
        for (int i = 0; i < zahlen.length; i++) {
            System.out.println("Index " + i + ": " + zahlen[i]);
        }
    }
```

———

```
/**
 * Generates and prints an array of random integers.
 * This method creates an integer array of the specified size ,
 * fills it with random numbers between 0 (inclusive) and 100 (exclusive),
 * and prints each value along with its index to the console.
 *
 * @param num the number of random integers to generate; must be ≥ 0
 */
```

Practice: Array

Write a method according to this documentation comment

```
/**
 * This method searches for a specific value in an integer array and returns its index.
 * If the value is found, it returns the index of the first occurrence of the value.
 * If the value is not found, it returns -1.
 *
 * @param array The array in which to search for the value.
 * @param value The value to search for in the array.
 * @return The index of the first occurrence of the value in the array, or -1 if the value is not found.
 */
```

Practice: array

Implement the sum method in ForLoop.

```
----------------------public class ForLoop
{
   /**
    * Your assignment is to implement this method:
    * Sum up every element in the array passed as
    * an argument and return the sum
    *
    * @param  a   an array of integers
    * @return     the sum of of all ints in a
    */
   public int sum(int[] a)
   {
      return 0;
   }
}
```

Geschweifte Klammern — Windows

Geschweifte Klammern {} kannst du in **Windows** einfügen, indem du gleichzeitig die Tasten **[Alt Gr] und [7]** bzw. **[0]** drückst.

- { = [Alt Gr] + [7]
- } = [Alt Gr] + [0]

Die Taste **[Alt Gr]** findest du rechts neben der Leertaste. Statt dieser Taste kannst du aber auch die Tastenkombination **[Strg] + [Alt]** verwenden.

Geschweifte Klammern Windows Tastatur

Practice

Given a 2D array board and a cell board [i][j].

List all of the cells that are its neighbors.

[i-1][j-1]	**[i-1][j]**	**[i-1][j+1]**
[i][j-1]	**[i][j]**	**[i][j+1]**
[i+1][j-1]	**[i+1][j]**	**[i+1][j+1]**

Practice: 2D Array

```
/**
 * Calculates the sum of all the elements in a specified column of a 2D array.
 * <p>
 * This method iterates through each row of the 2D array and adds the element
 * at the given column index to the sum. The column index is validated to ensure
 * that it is within the bounds of the array.
 * </p>
 *
 * @param columnIndex The index of the column to sum up. Must be a valid column index for the given 2D array.
 * @return The sum of all elements in the specified column. If the column index is out of bounds, it will return -1.
 */
public int sumColumn(int columnIndex)
{
}
```