14 Summary - Introduction

- Java vs. C/C++
  - ◆ no preprocessor, strongly typed, no pointer arithmetic, no procedures, ...
- Abstraction and Encapsulation
- Objects - Classes - Methods - Variables
- Constructors
- Packages: package, import
- Encapsulation: private, protected, public
- Applets and Applications

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

15 Overview of todays tutorial

- Overloading
- Inheritance
- Overriding
- static
- final
- abstract classes
- interfaces

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

16 Overloading

- The same method name can be used with different parameters
- Example:

```
class Date {
    ...
    void print(PrintStream stream) { stream.println(...); }
    void print() { print(System.out); }
}
```

- Notice: Overloading works only with parameters not return types

```
class Income {
    ...
    int computeIncome() { ... }
    float computeIncome() { ... } // Error !!
}
```

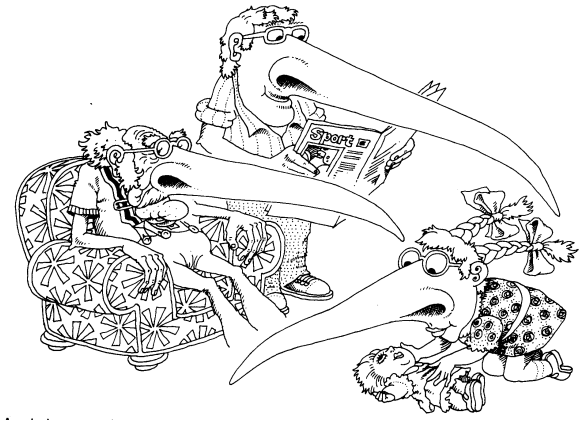

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17 Inheritance

- Define objects by referring to other objects
  - ◆ Example:
    - Definition of an animal: A thing that breathes and eats.
    - Definition of a cow: An animal that makes "muuuu" and gives milk.
    - Cow *inherits* the properties "breathes" and "eats" from animal.

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.1 Inheritance in the Real World



Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.2 Inheritance in Java

- Inheritance: Definition of a new object based on a specialized definition of an existing object.
- New classes can be derived from existing classes.
- Example: A customer is a person.

```
class Customer extends Person {
    int number;
    ...
}
```

- **Customer is a** *subclass* **of Person**

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.3 Subclasses

- Subclass inherits
  - State (instance variables) and
  - Behavior (methods) from superclass
- Subclass can
  - add new instance variables
  - add new methods
  - override inherited methods
  - shadow inherited instance variables (be careful!)

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

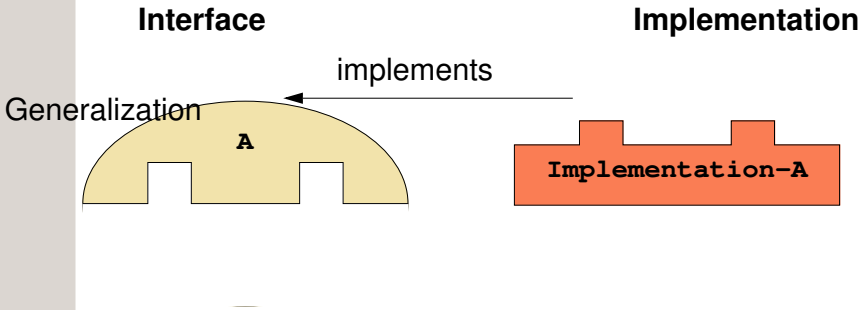
17.4 The Substitutability Principle

- An object of a subclass can be used wherever an object of the superclass is expected.
  - most important kind of polymorphism
- Inheritance is specialization ("is-a" relation, "kind-of" relation)
- Everything that holds for the generalization must also be true for the specialization.
  - The specialization fulfills all promises of the generalization.

→ Abstraction

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

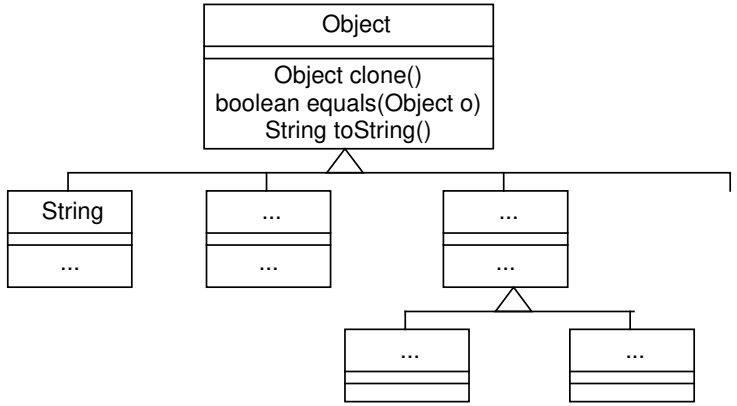
17.5 Inheritance is Specialization



Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.6 Inheritance in Java

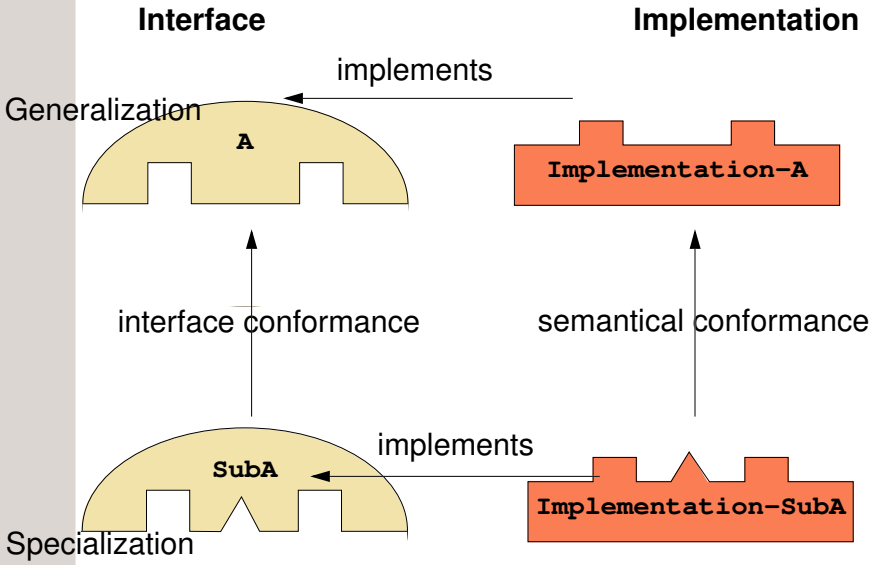
- Classes with single inheritance
- Tree hierarchy with class **`Object`** as base class of all other classes



- primitive types (**`int`**,**`float`**,...) are outside the class tree

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.5 Inheritance is Specialization



Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.7 Object

- Class **`Object`**: Base class of all classes

```
class Person extends Object {... }
```

  - **`extends Object`** can be omitted

```
class Person {... }
```

- Supplies some base functionality, for example:
  - **`boolean equals(Object o)`** // test for equality
    - default implementation compares references
    - you should provide your own implementation
  - **`String toString()`** // represent object as String
    - contains class name and object ID
    - you should provide your own implementation

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.8 Overriding

- Subclasses can provide a new implementation for an inherited method
- The new implementation *overrides* the inherited implementation.

```
class Person {
    String name;
    ...
    void print() {
        System.out.println("Person: " + name);
    }
}
class Customer extends Person {
    int number;
    ...
    void print() {
        System.out.println("Person: " + name);
        System.out.println("Customer number: " + number);
    }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.9 Overriding

- Invoke super class implementation with **`super.method()`**

```
class Customer extends Person {
    int number;
    ...
    void print() {
        super.print();
        System.out.println("Customer number: " + number);
    }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.10 Overloading vs. Overriding

- To override a method the parameter and return types must be identical (*no-variance*) otherwise the method is overloaded
- To fulfill the substitutability principle it would be sufficient that the parameters of the overriding method be supertypes (*contra-variance*) and return types are subtypes (*co-variance*) -- this is not supported by Java
- Frequent mistake:

```
class Object {
    boolean equals(Object o) { ... }
    ...
}

class Customer {
    int number;
    boolean equals(Customer c) { return number == c.number;}
    ...
}
```

does not *override equals of Object*

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.11 Dynamic Binding

- Method code is not linked until method invocation
- *Dynamic type*: type/class of the referenced object (class used with **`new`**)
- *Static type*: type of the reference
- The static type determines what methods can be invoked.
- The dynamic type is used to select the method.

```
Customer c = new Customer("Max", 1234);
Person p = c;  // dynamic type of p is Customer,
               // static type is Person

c.print();
p.print(); // although the reference p is of type Person,
           // the print() method of Customer is used
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.12 Visibility and Inheritance

- The visibility of methods must not be restraint in subclasses (substitutability).

```
class Person {
    public String getName() { ... }
}
class Customer extends Person {
    private String getName() { ... }
}
```

Error

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.13 Constructors and Inheritance

- Constructors are **not** inherited
- Invocation of the super-class constructor with `super(...)`
- `super(...)` must be the first statement in a constructor
- if first statement is not `super(...)`, the compiler automatically inserts a `super()` statement
- Default constructor
  - ◆ created by the compiler, if *no* constructor is defined
  - ◆ consists of `super()` - call

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.13 Constructors and Inheritance

- Example

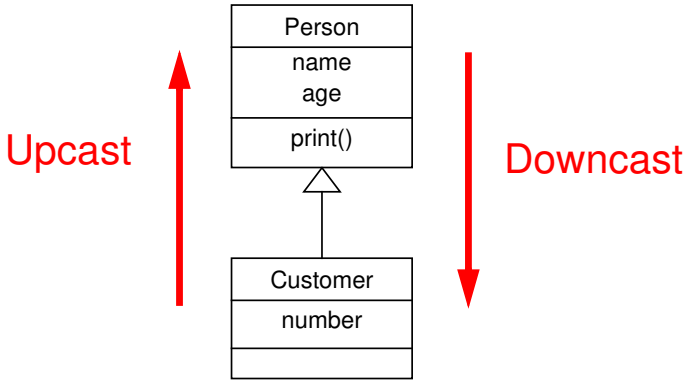
```
class Customer extends Person {
    int number;
    Customer(String name, int number) {
        super(name);
        this.number = number;
    }
    ...
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.14 Type Conversions: Upcast & Downcast

- Type conversion from subclass to superclass (*upcast*) is done automatically

```
Person p = new Customer(...);
```



Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.15 Downcast

- Type conversion from superclass to subclass (*downcast*) must be done with the cast operator

```
Customer c = new Customer(...)
Person p = c; // implicit type conversion
Customer c2 = (Customer) p;//explicit type conversion
```

- A **ClassCastException** is generated if object and variable are not type conform

```
Customer c = new Customer(...)
Person p = c; // implicit type conversion
Employee e = (Employee) p;
                  // generates a ClassCastException at run time
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.16 Type Deduction

- **instanceof** operator

```
Customer c = new Customer(...)
Person person = c; // upcast
if (person instanceof Employee) {
    Employee employee = (Employee) person;
    ...
} else if (person instanceof Customer) {
    Customer customer = (Customer) person;
    ...
}
```

- **instanceof** may be used with a superclass of the dynamic type

```
person instanceof Person
```

... would give **true** in the example

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.17 Class

- Class **Class**: The class of all classes.
- A Class object represents one class or interface
- Can be used to create new instances

```
Customer c = new Customer();
...
Class aClass = c.getClass();
System.out.println("Class of c is:"+aClass);

Object o = aClass.newInstance();  // now we have a new Customer
                                  // object
Person p = (Person) o;
```

- A class object can be created from a class name

```
Class aClass = Class.forName("Employee");
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

17.18 Class Literals

- since Java 1.1 there are class literals
- better than **Class.forName(...)**

```
Class aClass = Employee.class;
...

Object o = aClass.newInstance();
Person p = (Person) o;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

18 Other Language Features

- Class Methods/Variables
- Class Constructors
- Constants
- Final Methods
- Final Classes

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

18.1 Class Variables and Class Methods

- Classes can contain variables and methods: static variables and methods
- they can be used without an object
- Example:

```
class Test {
    private static int counter = 0;
    public Test() { counter++; }
    public static int howMany() { return counter; }
}
Test t = new Test();
System.out.println("No. of Test objects: " + Test.howMany());
```

- Example:
  - **`System.out`** is a static variable of the **`System`** class
  - **`System.out`** is of type **`PrintStream`**, **`PrintStream`** objects have **`println`** method

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

18.2 Class Constructors

- Classes are stateful (**`static`** variables) and must be initialized with a class constructor
- Example:

```
class Test {
    static int counter;
    static {
        counter = 9;
    }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

18.3 Constants

- Variables can be constant (*final*):
  - They must be initialized during declaration (JDK 1.0) or later (JDK 1.1: *blank finals*)
  - This initial value can not be changed.

```
class Test {
    public static final int x=5; // constant class variable
    private final int t=10;      // constant instance variable
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

18.4 Constants

- since Java 1.1 method parameters and local variables can be constant

```
class Test {
    String name;
    void setName(final String name) {
       final int i = 42;
       this.name = name;
    }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

18.5 Final Methods

- If methods are declared **final** they cannot be overridden
  - Security
  - Efficiency (makes inlining possible)

```
class Test {
    final void hello() {...}
}

class Test2 extends Test {
    void hello() { ... } // Error!! hello is final in Test
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

18.6 Final Classes

- If classes are declared **final** you cannot derive subclasses from them.
- Example:

```
public final class Test {
    ...
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

19 Abstract Classes

- Problem:
  - drawing editor with geometrical shapes (circle, rectangle,line)
  - Drawing sheet **Sheet** should only cope with **Shape**
  - **Sheet** must call **draw()** at **Shape**
  - **Shape** cannot provide a useful implementation of **draw()**
- **Shape** is an *abstract* class

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

19.1 Abstract Classes

- used to distill common properties of classes
- can **not** be used to create objects
- abstract classes contain not implemented methods (*abstract methods*)
- abstract classes and methods are declared with the **`abstract`** keyword

```
abstract class Shape {
    public abstract void draw();
}
```

- if a concrete class extends an abstract class it must implement all abstract methods

```
class Circle extends Shape {
    public void draw() { ... }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

20 Interfaces

- Java separates the class concept from the type concept
- Interfaces represent types
- Interfaces contain
  - ◆ method names and signatures
  - ◆ constants (**`static final`**)
- all methods are (implicitly) abstract
- classes can be compatible to interfaces (keyword **`implements`**)
- These classes must implement all methods of the interface.
- Definition of interfaces with the keyword **`interface`**

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

20.1 Example

1. Definition of an interface

```
public interface Printable {
    public void print();
}
```

2. Definition of classes that implement the interface

```
public class Account implements Printable {
    ...
    public void print() {
        System.out.println("balance="+balance());
    }
}
public class Person implements Printable { ... }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

20.1 Example

3. Using the interface

```
public class PrintQueue {
    public void add(Printable p ) { ... }
}
....
PrintQueue queue = new PrintQueue();
Printable p=new Person(...);
queue.add(p);
Account account = new Account(...);
queue.add(account);
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

20.2 Inheritance of Interfaces

- Interface can *inherit* multiple interfaces,
  classes can *implement* multiple interfaces
- Example:

```
interface Streamable extends FileIO, Printable {
    // additional methods
}

class Test implements Streamable, TestInterface {
    ...
}

class Test1 extends Test {
    ...
}
// Test1 is compatible to FileIO, Printable, Streamable,
// TestInterface and the class Test
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

21 Abstract Classes vs. Interfaces

- Abstract classes can provide a "partial" implementation of an abstraction.
- Abstract classes can contain instance variables.
- An abstract class should be used when only some parts of the implementation are "left open".
- An interface is suitable to represent certain properties (Printable, Clonable,...)

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

22 Inner Classes

- **local inner class**: only usable from enclosing class
- **inner class with method scope**: only usable in method
- **anonymous inner class**: only usable when defined
- **static inner class**: globally usable

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

22.1 Local Inner Classes

- Inner class can access instance variables of enclosing class

```
class Test {
  private String array[] = { "hans", "otto", "max"};
  class Inner {
    String method(int i) { return "Name:+" + array[i]; }
  }
}
```

- visibility modifiers like methods and instance variables
  (`private`,default,`protected`,`public`)

```
class Test {
  private String array[] = { "hans", "otto", "max"};
  private class MyEnum implements Enumeration {
    private int counter = 0;
    public Object nextElement() { return [counter++]; }
    public boolean hasMoreElements() { return counter<array.length; }
  }
  public Enumeration enumerate() { return new MyEnum(); }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

22.2 Inner Classes with Method Scope

- class only needed in one method

```
class Test {
  private String array[] = { "hans", "otto", "max"};
  public Enumeration enumerate() {
    class MyEnum implements Enumeration {
      private int counter = 0;
      public Object nextElement() { return [counter++]; }
      public boolean hasMoreElements() { return counter<array.length; }
    }
    return new MyEnum();
  }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

22.2 Inner Classes with Method Scope

- Can access **final** parameters and **final** local variables of the enclosing method
- Access to **this** of the enclosing class **X** with **X.this**

```
class Test {
  public void test(final String msg) {
    class Inner {
     public void output(String hello) { System.out.println(hello+msg);}
    }
    ...
  }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

22.3 Anonymous Classes

- Class name **MyEnum** in previous example contains no information ➜ Use an anonymous inner class

```
class Test {
  private String array[] = { "hans", "otto", "max"};

  Enumeration enumerate() {
    return new Enumeration() {
      int counter = 0;
      public boolean hasMoreElements() {
        return counter < array.length; }
      public Object nextElement() {
        return array[counter++]; }
    };
  }
}
```

concludes return statement

Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.

22.4 Static Inner Classes

- Static classes can only access static variables and methods of the enclosing class

```
class Test {
  private static String array[] = { "hans", "otto", "max"};

  static class E implements Enumeration() {
    int count = 0;
    public boolean hasMoreElements() {
      return count < array.length; }
    public Object nextElement() {
      return array[count++]; }
  }
}
...
Enumeration e = new Test.E();
System.out.println(e.nextElement());
```


Reproduktion jeder Art oder Verwendung dieser Unterlage bedarf der Zustimmung des Autors.