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
15 Overview of todays tutorial

- Overloading
- Inheritance
- Overriding
- static
- final
- abstract classes
- interfaces
The same method name can be used with different parameters

Example:

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

Notice: Overloading works only with parameters not return types

```java
class Income {
    ...
    int computeIncome() { ... }
    float computeIncome() { ... } // Error !!
}
```
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.
17.1 Inheritance in the Real World
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.

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

Customer is a *subclass* of Person
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!)
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
17.5 Inheritance is Specialization

Interface

Generalization

A

Implements

Implementation

Implementation-A
Inheritance is Specialization

Interface

Generalization

A

implements

interface conformance

Specialization

SubA

implements

Implementation

Implementation-A

semantical conformance

Implementation-SubA
17.6 Inheritance in Java

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

```
Object
  \__ Object clone()
  \__ boolean equals(Object o)
  \__ String toString()
```

primitive types (int, float, ...) are outside the class tree
17.7 Object

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

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

- **extends Object** can be omitted

```java
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
### 17.8 Overriding

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

```java
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);
    }
}
```
17.9 Overriding

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

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

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

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

*does not override equals of Object*
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.

```java
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
               // the print() method of Customer is used
```
Visibility and Inheritance

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

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

Error
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
17.13 Constructors and Inheritance

Example

```java
class Customer extends Person {
    int number;
    Customer(String name, int number) {
        super(name);
        this.number = number;
    }
    ...
}
```
17.14 Type Conversions: Upcast & Downcast

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

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

![Diagram showing inheritance with upcast and downcast examples]
17.15 Downcast

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

```java
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

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

- **instanceof** operator

```java
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

```java
person instanceof Person
```

... would give `true` in the example
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

```java
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

```java
Class aClass = Class.forName("Employee");
```
since Java 1.1 there are class literals

better than `Class.forName(...)`

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

Object o = aClass.newInstance();
Person p = (Person) o;
```
18 Other Language Features

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

OODS © 1997-2000 Michael Golm
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:

```java
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
18.2 Class Constructors

- Classes are stateful (static variables) and must be initialized with a class constructor.

- Example:

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

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

since Java 1.1 method parameters and local variables can be constant

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

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

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

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

- If classes are declared `final` you cannot derive subclasses from them.

- Example:

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

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

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

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

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

1. Definition of an interface

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

2. Definition of classes that implement the interface

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

public class Person implements Printable {
    ...
}
```
3. Using the interface

```java
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);
```
20.2 Inheritance of Interfaces

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

Example:

```java
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
```
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,...)
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
22.1 Local Inner Classes

- Inner class can access instance variables of enclosing class

```java
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)

```java
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(); }
}
```
22.2 Inner Classes with Method Scope

- class only needed in one method

```java
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();
    }
}
```
### 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

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

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

```java
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*
22.4 Static Inner Classes

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

```java
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());
}
```