Secure OOP with Java

Lecture Unit - 04

Claudia Maderthaner <claudia.maderthaner@fh-hagenberg.at>

Inheritance

Definition

Inheritance is a mechanism for propagating properties (attributes and methods) of superclasses to subclasses.

Characteristics

- Creating a new class by reusing code from an existing class
  - Fields
  - Methods
- The subclass inherits
  - Specification
  - Implementation

Declaring a Subclass

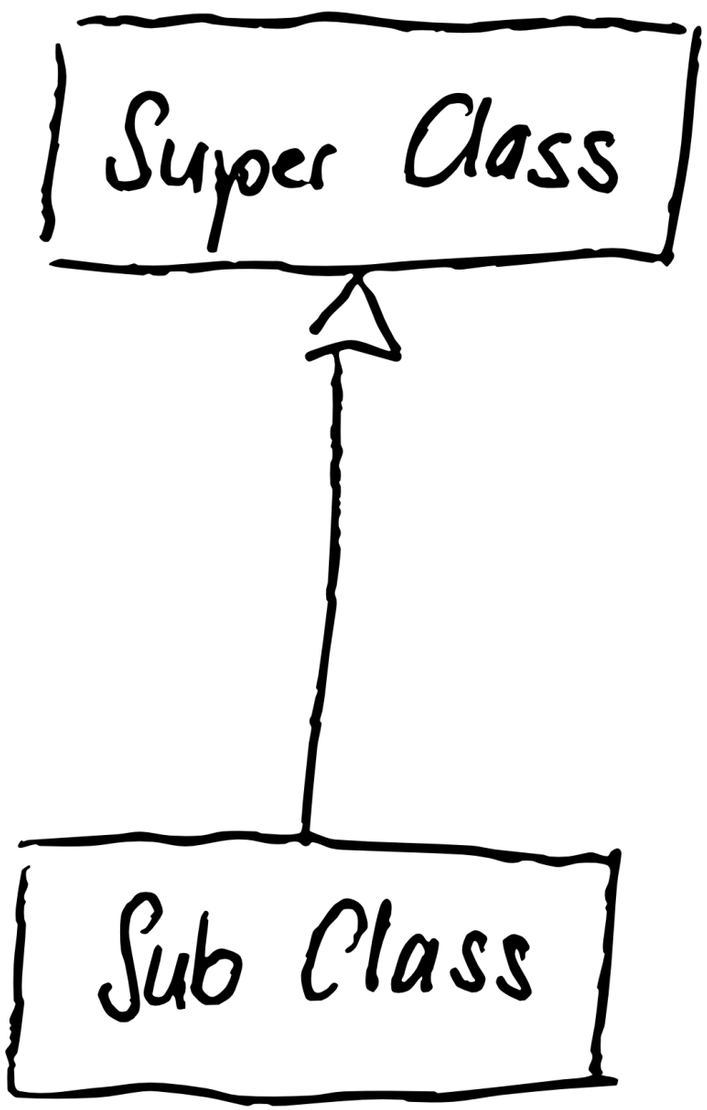
```
[modifiers] class SubClass extends SuperClass {
  // additional fields and methods
}
```

```java
public class Person {
  String firstname;
  String lastname;

  public String getFullname() {
    return firstname + " " + lastname;
  }
}
```

```java
public class User extends Person {
  String login;

  public String getLogin() {
    return login;
  }
}
```



base class
parent class

derived class
child class

Specification

The subclass inherits

- the interface (API) and
- the data type

of the superclass.

Implementation

The subclass

- also inherits the implementation but
- may override the method implementations and
- can add additional fields and methods.

```
public class User extends Person {
  String login;

  public String getFullname() {
    return firstname + " " + lastname + " (" + login + ")";
  }
}
```

super Constructor

- Calls the constructor of the superclass
- Has to be the first statement in the constructor
- It is not possible to combine `super` and `this` calls in a single constructor

super Constructor and Inheritance

> There must be a call to the constructor of the super class.

- If there is no explicit call of the `super` constructor the compiler adds the call of the default constructor of the superclass
  - If there is no default constructor in the superclass compilation fails and an explicit call of a superclass constructor has to be added
- When calling the superclass constructor in the code any constructor (with or without parameters) might be called

Subclass Constructors

```
public class Publication {
    String name;

    public Publication() {
        // no op
    }

    public Publication(String name) {
        this.name = name;
    }
}
```

```
public class Book extends Publication {
    String author;

    public Book() {
        // no op
    }
}
```

Subclass Constructors

```
public class Publication {
    String name;

    public Publication(String name) {
        this.name = name;
    }
}
```

```
public class Book extends Publication {
  String author;

  public Book(String name, String author) {
      super(name);
      this.author = author;
  }
}
```

Method Overriding

Redefining an instance method, which is inherited from the superclass

Rules for Method Overriding

- The method must be an instance method.
- The overriding method must have the same signature as the overridden method.
- The return type of the overriding and the overridden methods
  - must be the same for primitive data types,
  - must be assignment-compatible with the return type of the overridden method for reference data types.

Rules for Method Overriding

- The access modifier of the overriding method must be at least the same or more relaxed than that of the overridden method.
- The method cannot add a new exception to the list of exceptions in the overridden method.

Method Overriding	**Method Overloading**
Involves inheritance and at least two classes	Involves only one class
Only occurs for methods with same signature	Only occurs for methods with same name but different parameter list
`return` type has to be assignment compatible	`return` type is of no consequence
No additional throws clauses possible, may have same or less restrictive list of exceptions	Throws clauses are of no consequence
Only applies to instance methods	Applies to any method (static and non-static)

@Override

- Documents that a method overrides a method in a superclass.
- Compiler and IDEs may check whether there is a method with the same signature in the class hierarchy.

```
1 public class User extends Person {
2     String login;
3
4     @Override
5     public String getFullname() {
6         return firstname + " " + lastname + " (" + login + ")";
7     }
8 }
```

Accessing an Overridden Methods

```
public class User extends Person {
    String login;

    @Override
    public String getFullname() {
        return super.getFullname() + " (" + login + ")";
    }
}
```

Method Hiding

There is no overriding for static methods.

If a static method is redefined in a subclass the subclass implementation hides the superclass implementation.

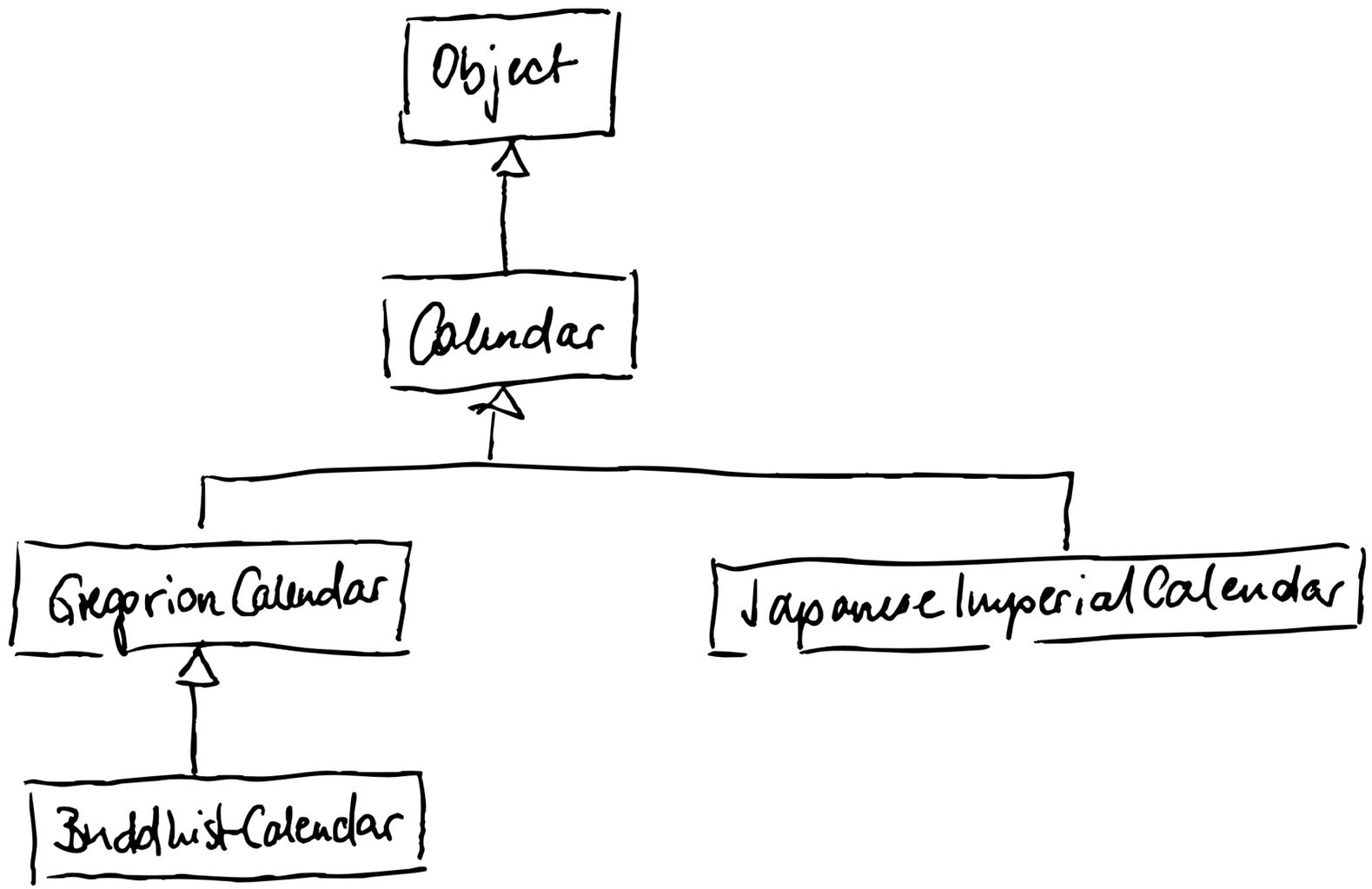
Variable Hiding

There is no overriding for fields. Do not re-declare already existing fields.

```
public class Person {
    String firstname;
    String lastname;
}
```

```
public class User extends Person {
    String firstname;
    String lastname;
    String login;
}
```

Inheritance Hierarchies



Inheritance Chain

```
public class Animal {
}
```

```
public class Mammal extends Animal {
}
```

```
public class Dog extends Mammal {
}
```

Multiple Inheritance

Java does not support multiple inheritance for classes.

Final Classes

```
public final class Human {

}
```

It is not possible to use final classes as a superclass.

Sealed Classes

Sealed classes and interfaces restrict which other classes or interfaces may extend or implement them.

Declaring a sealed class

```
public sealed class Vehicle permits Car, Truck {
}
```

```
public non-sealed class Car extends Vehicle {
}
```

```
public final class Truck extends Vehicle {
}
```

Constraints on Permitted Subclasses

- They must be accessible by the sealed class at compile time.
- They must directly extend the sealed class.
- They must have von of the following modifiers:
  - `final` - class cannot be extended
  - `sealed` - class itself is a sealed class
  - `non-sealed` - class allows any subclasses

Casting

- Instances of a subclass can be used everywhere the superclass can be used (no explicit cast necessary).
- Variables of the superclass must be cast to be used for a subclass variable.

instanceof

- Checks whether an object is assignable to a specific data type
- Returns a `boolean` (`true` or `false`)

```
User user = new User();
Person person = user;
```

```
Person person = new User();
if (person instanceof User) {
    User user = (User) person();
}
```

Object

java.lang.Object

- `Object` is the root of the Java class hierarchy.
- Every class has `Object` as a superclass.
  - When a class is missing an explicit superclass, `Object` is implicitly used as superclass.
- All objects, including arrays, implement the methods of this class.

The equals Method

- Tests whether one object is considered equal to another.
    - Each class has to determine what equal means

The `equals` implementation in `Object` compares the object references.

equals Contract

reflexive

`x.equals(x)` must be `true`

symmetric

`x.equals(y)` must be `true` if and only if `y.equals(x)` is `true`

transitive

if `x.equals(y)` is `true` and `y.equals(z)` is `true` then `x.equals(z)` must be `true`

consistent

multiple invocation of `x.equals(y)` must consistently return `true` or `false` given the object states are not changed

Every object is not equal to `null`.

equals Checklist

1. check whether it is the same object
2. check whether it is the same type
3. cast to the type
4. check whether all relevant fields are equal
5. implement `hashCode()` accordingly

```java
public class Type {
    private int attribute;

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof Type)) {
            return false;
        }
        Type other = (Type) obj;
        if (attribute != other.attribute) {
            return false;
        }
        return true;
    }
}
```

The hashCode Method

- A hash code is an integer that is derived from an object.

hashCode "Recipe"

1. Declare a variable `result` of type `int` and initialize with `1`
2. For each relevant field in the class
    - calculate the hash code `c` and
    - update the result `result = 31 * result + c`
3. Return the overall result

The toString Method

- Returns a String representing the value of the current object.
- Every class should offer a meaningful implementation of `toString`.
  - The debugger makes use of `toString`, to render current variable states.

Interfaces

Characteristics

- A set of requirements for the classes that want to conform to the interface.
- Describe what classes should do, without specifying how they should do it.
- Interfaces are also data types.

Declaring an Interface

```
[modifiers] interface InterfaceName {
  DataType CONSTANT_NAME = value;

  [modifiers] ReturnType methodName(ParameterType... param);
}
```

It is not possible, to declare fields in interfaces.

```
public interface Comparable {
    int compareTo(Object o);
}
```

```
public interface Printable {
    String DEFAULT_ENCODING = "UTF-8";

    void print(Object o, String encoding);
}
```

Interface Documentation

```
 * }</pre>
 *
 * It follows immediately from the contract for {@code compareTo} that the
 * quotient is an <i>equivalence relation</i> on {@code C}, and that the
 * natural ordering is a <i>total order</i> on {@code C}.  When we say that a
 * class's natural ordering is <i>consistent with equals</i>, we mean that the
 * quotient for the natural ordering is the equivalence relation defined by
 * the class's {@link Object#equals(Object) equals(Object)} method:<pre>
 *     {(x, y) such that x.equals(y)}. </pre><p>
 *
 * In other words, when a class's natural ordering is consistent with
 * equals, the equivalence classes defined by the equivalence relation
 * of the {@code equals} method and the equivalence classes defined by
 * the quotient of the {@code compareTo} method are the same.
 *
 * <p>This interface is a member of the
 * <a href="{@docRoot}/java.base/java/util/package-summary.html#CollectionsFramework">
 * Java Collections Framework</a>.
 *
 * @param <T> the type of objects that this object may be compared to
 *
 * @author  Josh Bloch
 * @see java.util.Comparator
 * @since 1.2
 */
public interface Comparable<T> {
    /**
     * Compares this object with the specified object for order.  Returns a
     * negative integer, zero, or a positive integer as this object is less
     * than, equal to, or greater than the specified object.
     *
     * <p>The implementor must ensure {@link Integer#signum
     * signum}{@code (x.compareTo(y)) == -signum(y.compareTo(x))} for
     * all {@code x} and {@code y}.  (This implies that {@code
     * x.compareTo(y)} must throw an exception if and only if {@code
     * y.compareTo(x)} throws an exception.)
     *
     * <p>The implementor must also ensure that the relation is transitive:
     * {@code (x.compareTo(y) > 0 && y.compareTo(z) > 0)} implies
     * {@code x.compareTo(z) > 0}.
     *
     * <p>Finally, the implementor must ensure that {@code
     * x.compareTo(y)==0} implies that {@code signum(x.compareTo(z))
     * == signum(y.compareTo(z))}, for all {@code z}.
     *
     * @apiNote
     * It is strongly recommended, but <i>not</i> strictly required that
     * {@code (x.compareTo(y)==0) == (x.equals(y))}.  Generally speaking, any
     * class that implements the {@code Comparable} interface and violates
     * this condition should clearly indicate this fact.  The recommended
     * language is "Note: this class has a natural ordering that is
     * inconsistent with equals."
     *
     * @param   o the object to be compared.
     * @return  a negative integer, zero, or a positive integer as this object
     *          is less than, equal to, or greater than the specified object.
     *
     * @throws NullPointerException if the specified object is null
     * @throws ClassCastException if the specified object's type prevents it
     *         from being compared to this object.
     */
    public int compareTo(T o);
}
```

Implementation of Interfaces

- A class may implement zero, one or more interfaces.
- Different classes may implement the same interface in a different way.

```
public class Human implements Comparable {
  String name;

  int compareTo(Object o) {
    if (o instanceof Human) {
      return name == null ? return 1 : name.compareTo(o.name);
    }
    return -1;
  }
}
```

Extending Interfaces

- An interface may extend one or more interfaces.

Interfaces as a Contract

- Interfaces form a contract between the class and the outside world
- This contract is enforced at build time by the compiler.
- Classes implementing an interface must provide implementations for the methods declared in the interface.

Interfaces as Data Types

- Interfaces are not classes. There is no constructor and no instances of the interface itself.
- But it is possible to declare variables with the interface as data type. → Every instance of a class implementing the interface may be referenced from this variable.

default Methods

- Since Java 8 interfaces may contain methods with default implementations.

```
public interface Greeter {
    String getName();

    default void greet() {
      System.out.println("Hello " + getName() + "!");
    }
}

public class Human implements Greeter {
  String name;

  public Human(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

```
Human human = new Human("John");

human.greet(); // -> Hello John!
```

Diamond Problem

When a method is defined in a superclass and as a default method in one or more implemented interfaces, the following rules apply:

1. Superclasses win → default methods with the same signature are ignored
2. Interfaces clash → if there is a default method and another interface contains a method with the same signature, the conflict must be resolved by overriding this method

static Methods

- Since Java 8 interfaces may contain static methods.

```
public interface Path {
    public static Path of(URI uri) {
        // ...
    }

    public static Path of(String first, String... more) {
        // ...
    }
}
```

private Methods

- Since Java 9 interface may implement private methods.
- Private methods in interfaces may not be abstract.
- They may only be used within the interface code.

Abstract Classes

Characteristics

- Abstract classes may declare methods with no implementation.
- Therefore, it is not possible to create instances of abstract classes.

Declaring an Abstract Class

```
[modifiers] abstract class AbstractClass {
  // additional fields and methods
}
```

Rules for Abstract Classes

- A class may be declared `abstract` even if it does not have an `abstract` method.
- A class must be declared `abstract` if
  - it declares an `abstract` method,
  - it inherits an `abstract` method and does not provide an implementation, or
  - it implements an interface and does not provide an implementation for all methods declared in the interface.

Rules for Abstract Classes

- You cannot create an instance of an abstract class.
- An abstract class cannot be declared `final`.
- An abstract class must provide at least one non-private constructor.
- An abstract method cannot be declared `static` or `private`.

Missing method

```
public abstract class Human implements Comparable {
  String name;

  // missing implementation of interface method
}
```

Method without implementation

```
public abstract class Human {
  String name;

  public abstract void greet();
}
```

Contact

Moodle Discussion Board

claudia.maderthaner@fh-hagenberg.at