Object-Oriented Software Development

Advanced Principles

Faktorisierung: Sachverhalt nur an einer Stelle im SW System vorkommen
Statisch vs. dynamischer Typ: dynamischer erst zu Laufzeit bekannt, statischer Typ der Variablen ergibt sich durch Deklaration

Overview

Contents

- Type checking
-

Naming Conventions

- Classes
  - Nouns, first letter of each word is capitalized
  - Example: `ShadowedBox`
- Interfaces
  - Start with I, first letter of each internal word is capitalized
  - Example: `IMouseListener`
- Methods
  - First letter lowercase,
    first letter of each internal word is capitalized
  - Example: `setCoordinate`
- Instance variables
  - Starts with `f`, first letter of each internal word is capitalized
  - Example: `fFillPattern`

Naming Conventions

- Class variables
  - Starts with `s`, first letter of each internal word is capitalized
  - Example: `sMinSize`

- Constants
  - All uppercase, words are separated by "_"
  - Example: `MAX_ITEMS`

Object vs. Value

- Value
  - Describe abstract values
  - No lifecycle
  - No side effects (when shared)
  - Immutable state (conceptually),
    can only be interpreted (accessible by different representations)
  - Examples: integer, float, character, monetary value, account number

- Object
  - Describe a phenomenon in a problem domain
  - Can be distinctively referenced
  - Can be shared ➜ side effects
  - Lifecycle (creation, mutation, deletion)
  - Examples: account, payment, person

Object vs. Value

- Implementation has to follow object or value semantics!

- Implementation of value semantics:
  - Immutability
  - Copy on write

- Advantages: simple, easy sharing and threadsafeness

- Java
  - Class "Integer" is implemented with value semantics
    - Instances of Integer are immutable
    - Same holds true for BigInteger, BigDecimal etc
  - Class "Date" is implemented with object semantics
    - Mutable ➜ side effects
    - Date is deprecated and (partially) replaced by Calendar

Types and Classes

- Type: Abstract description of properties of a set of objects
- Class: Implementation of a type
  Object ➜ Class ➜ Interface ➜ Type).
- Static type checking
  - Class/interface is a type (e.g. in Java, C++,…)
  - Assignment compatibility only along the class/interface hierarchy
- Dynamic type checking
  - Protocol of an object = type
    - Protocol is the set of messages understood by an object
  - More flexible than static type checking
  - Assignment compatibility is not restricted by class hierarchy

Prototypes

- Some languages get by without classes, e.g. JavaScript
- Object creation in JavaScript

```
           =
            :          ,
           :            ,
          :          ()
                 .          +     +     .        ;
    ;
 ;
```

- Constructor definition in JavaScript

```
           =          (         ,         )
       .          =          ;
       .         =         ;
         .         .        =          ()
                 .          +     +     .        ;
    ;
 ;

            =           (         ,            );
```

Type Checking

- Static type checking
  - Type errors are usually detected at compile time
  - Allows for generating efficient target code (e.g. byte code)
  - Type errors can still occur at run-time (e.g. by type casting)

```
        =                ();
  .    ();
```

Compiler detects invalid method call

- Dynamic type checking
  - Flexibility at the expense of reliability
  - Developer is responsible for invoking valid methods

```
          =       A          <     >();
    .    ();
```

„MissingMethodException‟ at run-time

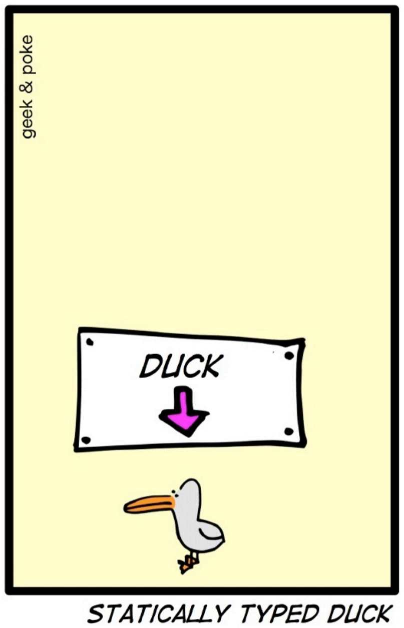
- **Dynamic type checking ≠ weak type checking**

Type Checking

- Static type checking ⇔ dynamic binding
  - Static type checking ensures that a message is understood by an object
  - Dynamic binding selects the appropriate method based on the dynamic type of the receiver

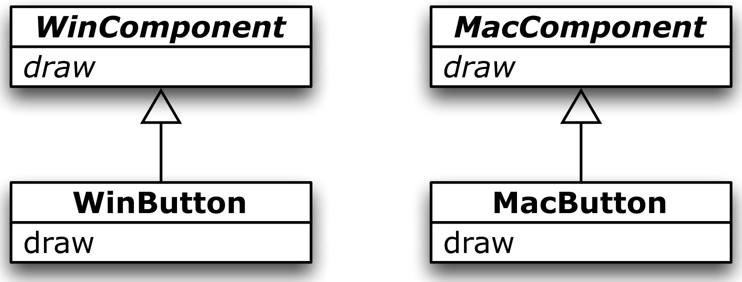
- Static type checking ⇔ inheritance
  - An object of type T' can be assigned to a variable of static type T where T' is derived from T
  - Assignment compatibility is restricted by type hierarchy
  - Dynamic binding is only possible along the type hierarchy

- Dynamic type checking (dynamic typing)
  - Allows for dynamic binding outside of the type hierarchy
  - Criterion: does an object understand a message?
  - Smalltalk developers: „True Dynamic Binding“
  - Groovy developers: “Duck Typing”
    - Looks like a duck, walks like a duck, quacks like a duck ➜ D ck



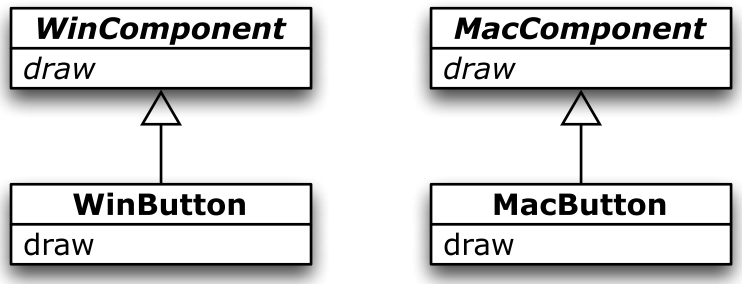
Type Checking

- Static type checking




```
C                ;
C                ;
B          =           B      ();

=    ;  //
=    ;  //C                      !
```

- Dynamic type checking



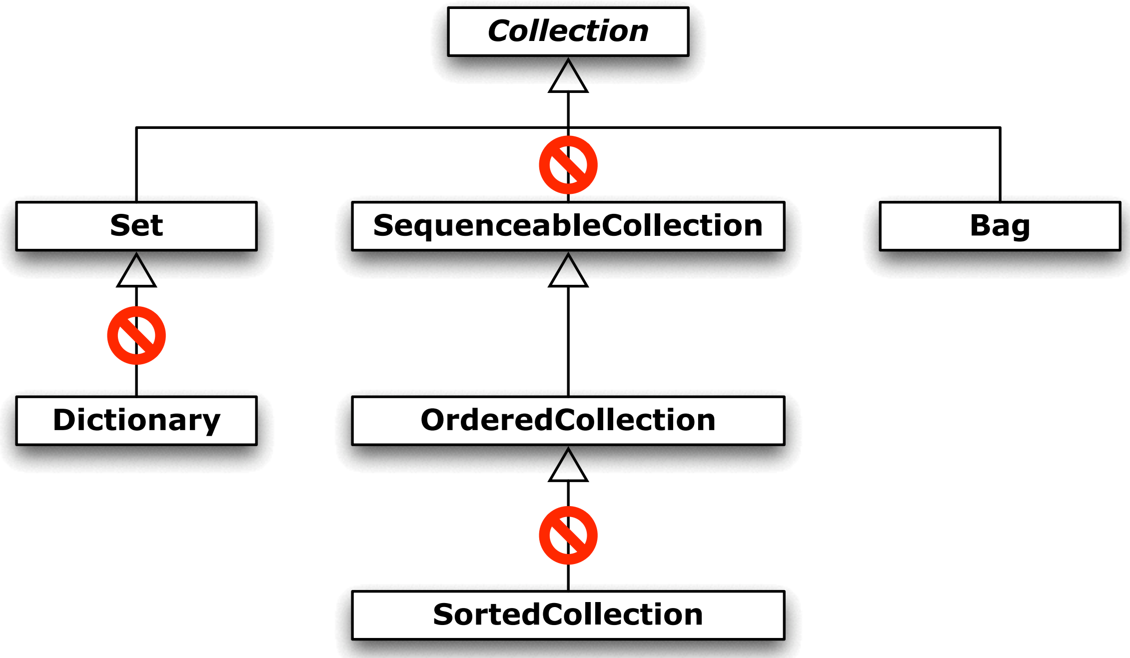

```
    =           B      ()

=       //
=       //
.    () //   B      .     ()
```

Subtype and Subclass

- Subtype: T‵ is a subtype of T if T-objects can be replaced by T‵-objects
  „Principle of Substitutability‟, Is-A-Relationship

- Subclass: T′ is a subclass of T if T′ is derived from T but T-objects cannot be replaced by T′-objects

- Example from the Smalltalk class library:



Principle of Substitutability – Requirements

- Subtype must provide at least the interface of its base type
  - Restrictions are not permitted!
- Pre- and postconditions
  - Preconditions must not be narrowed by a subtype
  - Postconditions must not be widened by a subtype
- Exceptions
  - Subtypes must not throw new exceptions
  - *Unless*: exception is a subtype of the exception thrown by the base type

Subtype and Subclass

- Interface inheritance:
  - Enables polymorphism
  - Modeling conceptual relationship among types

- Implementation inheritance:
  - Reusing the functionality of the base class
  - Code sharing

- Interface inheritance ➜ subtype
  Implementation inheritance ➜ „only" subclass

➜ Discrimination between subtype and subclass is essential in software design!

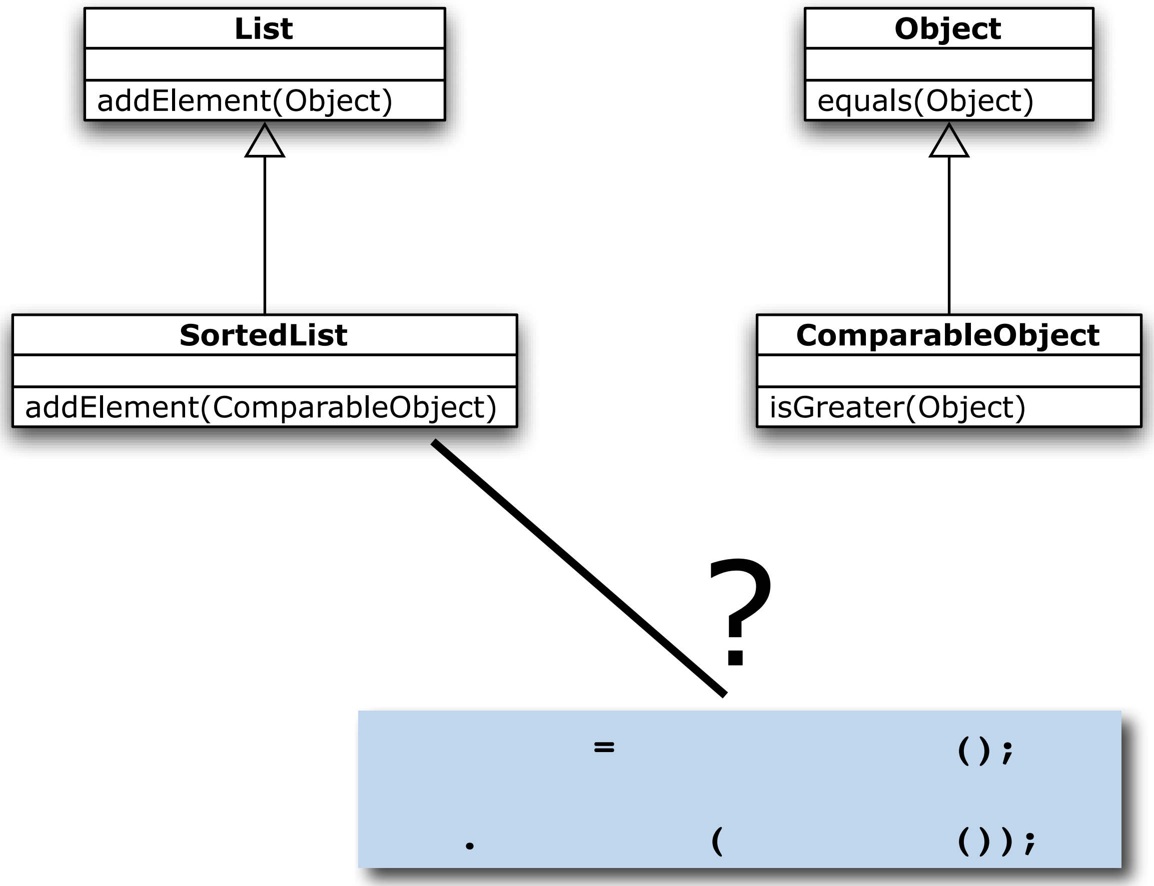
Subtype and Subclass

- Design goal:

Minimize implementation inheritance

➜ promotes comprehensiveness/maintainability of software

- Most object-oriented programming languages provide language features to differentiate between interface and implementation inheritance
- Examples: C++, Java, Eiffel


List
addElement(Object)
Object
equals(Object)
SortedList
addElement(ComparableObject)
ComparableObject
isGreater(Object)
?
= ();
. (());

Covariance and Contravariance

- Is the principle of substitutability applicable to method parameters in overriding methods?

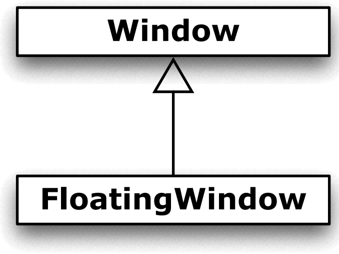
- *Covariance*: Type of a method parameter is narrowed in a subclass
  - Principle of substitutability applied to method arguments
  - Useful, but expensive to be checked statically!

- *Contravariance*: Type of a method parameter is widened in a subclass (or stays the same).

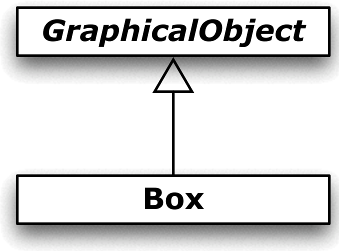
- Most object-oriented languages only support contravariance
  - Exception: Eiffel

Types of Inheritance

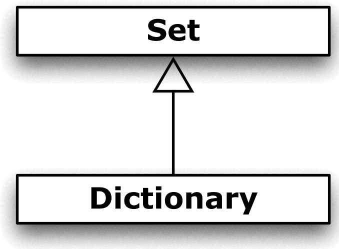
- Specialization
  - Derive concrete subclass from concrete base class



- Specification
  - Concrete subclass implements abstract base class

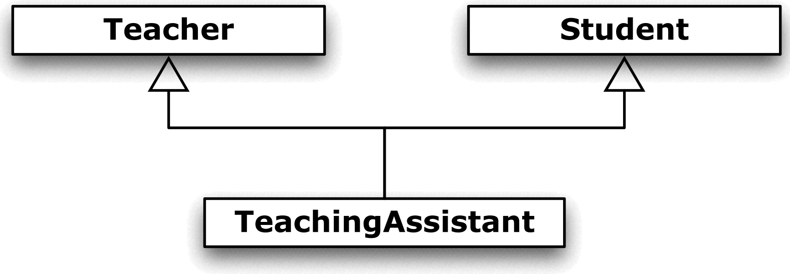


- Restriction
  - Subclass restricts the base class interface

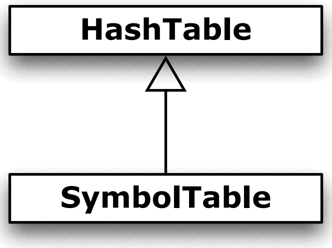


Types of Inheritance

- Combination
  - Multiple inheritance

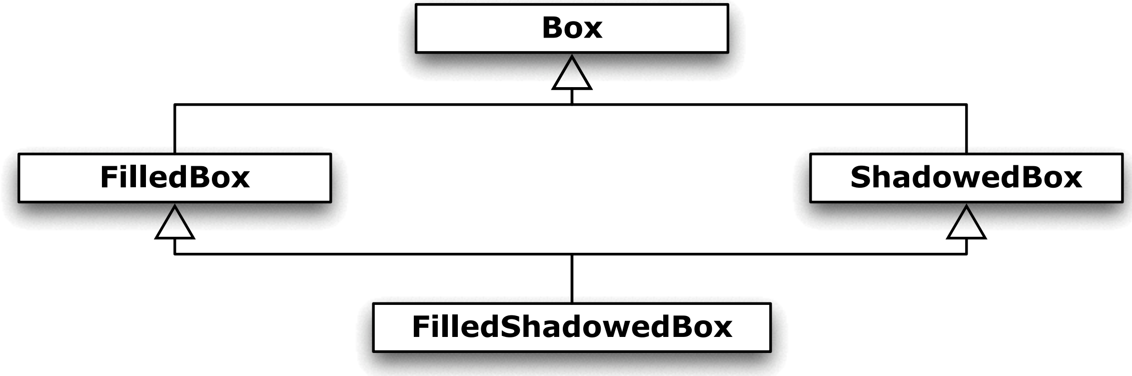


- Construction
  - Implementation inheritance



Multiple Inheritance

- Multiple inheritance from the same class? (Fork-Join)



- C++: FilledShadowedBox has two Box objects
- Virtual inheritance from Box:
  - A single Box object will be shared by all derived classes

```
      B    ;
            B  :                    B    ;
              B  :                    B    ;
                    B  :                  B    ,                     B    ;
```

- Virtual inheritance is conceptually „expensive“
  - ➜ Requires assumptions on future derived classes

Restricting the Inheritance Interface

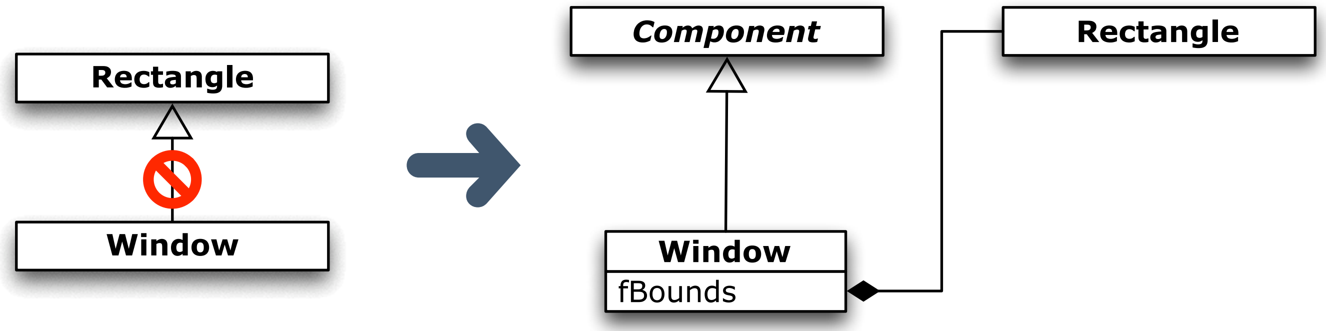
- Prevent method overriding
  - Java: keyword "final"
  - Turns off dynamic binding
    - Enables optimizations by the compiler or run-time system
  - Excludes alternative implementations
    - For example: export restrictions in cryptographic libraries

- Prevent derived classes
  - Java: keyword "final"
  - All methods are final in a final class
  - Excludes alternative implementations
  - Prohibit mutable derivations of immutable classes

Inheritance vs. Composition

- Inheritance
  - Protocol is extended (restriction contradicts subtyping)
  - Subtyping (is-a relationship)
    - Conforms to principle of substitutability!
  - Protocols of base class and subclass are very similar

- Composition
  - Little compliance between base class and subclass
    - Base class is just a minor detail of the subclass
  - Base class is a means of implementation (has-a relationship)
  - Implementation is substitutable



Design for Inheritance

- Inheritance violates encapsulation
  - Subclass needs to know implementation details
  - Judiciously grant access to subclasses by using protected access
  - By default members should be private
- Classes have to rely on the sanity of base classes and subclasses
- Constructors must not invoke overridable methods
- Test inheritability by writing subclasses!
- Prohibit subclassing if class cannot be safely subclassed
  - ➜ e nal
  - Imm able cla e m no be bcla ed

Java – Interfaces

- „Class‟ which declares only abstract methods
  - Interface defines a type, most abstract type specification
  - No state allowed (except constants)

- Class can inherit from 0-1 classes and implement 0-n interfaces

```
                    (          ,          );
```

```
                B          B
                    (          ,          )    ;
```

```
                         (          )    ;
```

- Use interfaces as static type declarations whenever possible!

Java – Interfaces

- Tagging (or marker) interfaces
  - Empty interface
  - Purpose:
    - Marking of a class to be eligible for being processed by some other class
    - Default implementation
  - Examples
    - java.lang.Cloneable
      - Instances (of classes which are marked as Cloneable) may be copied
    - java.io.Serializable
      - Instances may be serialized

- Constant interface
  - JDK example: java.io.ObjectStreamConstants
  - Implementation detail becomes part of the class’s exported API
  - Since Java 5: use static import instead

Abstract Classes

- Purpose:
  - Define common structural and functional properties
  - Do not allow for instance creation

- Factorization
  - Common properties of subclasses are defined in an abstract class
  - Inheritance allows to adapt resp. implement these properties
  - Implementation of a system can be based on abstract classes

- Ease the implementation of concrete classes
  - Several concrete methods can be based on a few abstract methods
  - Example: Comparison of objects
    All comparison methods can be based on "=" and ">"
    ➜ Small inheritance interface, broad method interface

- Facilitates comprehensibility of a class hierarchy

Abstract Classes in Java

- Abstract method
  - Only the signature is defined
  - Implementation is left to derived classes
  - Abstract method implies an abstract class
- Abstract class
  - Explicitly defined as an abstract class
  - Can contain 0-n abstract methods

```
                c          ;
                     (          );
         c             (            )   c           =   ;
                 c          ()          c          ;
```

Abstract Classes and Static Type Checking

- Smalltalk: Abstract classes are based on conventions (self subclassResponsibility)
- Java: Abstract classes are supported by the language

```
          B     ;
                (          );
      B   (           )  B       =  ;
           B     ()        B     ;

   =          ();
```

Compile time error

- Principle of Substitutability
  - *Static type checking*: Common (abstract) base class resp. interface
    ➜ documentation of design
  - *Dynamic type checking*: compatible method signature

Abstract Classes and Interfaces

- Interfaces should be as small as possible
  - All methods of an interface are abstract
  - Implementing class must implement all methods of its interfaces
  - Small interfaces ease implementability and substitutability

- Combining abstract classes and interfaces
  - Abstract class "implements" interface
  - Methods are not abstract but implementation is empty
  - Client may:
    - Implement interface ➜ forced to implement all methods
    - Inherit from abstract class ➜ override only required methods

Abstract Classes and Interfaces

- Example from the Java class library

```
C      (            );
       (            );
      (            );
       (            );
        (            );
```

```
       A
C      (            )   ;
       (            )   ;
      (            )   ;
       (            )   ;
        (            )   ;
```

Information Hiding

- Goal: abstracting from the implementation
  - Minimal coupling between classes
  - Implementation can be replaced
- Restricted access rights:
  - Discrimination between clients and inheritors
    - Java: public vs. protected
  - Classes within a subsystem
    - Java: public vs. package private
    - C++: friends
- Accessors hide state implementations (representation of state)
  - Advantages:
    - Minimized coupling, implementation can be replaced
  - Disadvantages
    - More verbose and inefficient (optimizations as in Java are feasible)

- Abstract state
  - Specification of state by abstract accessors in the abstract base class
  - Implementation of state in derived class (incl. concrete accessors)

```
                     A       B
                                 ();
                               (            );

  //
```

```
            B              A       B
                       ;

                        ()                ;
                      (            )          =      ;

  //
```

Inheritance and Information Hiding

- Two kinds of interfaces: clients and inheritors
  - What access rights are provided?

- Open-closed principle (B. Meyer)
  - Open: Class can be extended / extended in derived class
    ➜ inheritors
  - Closed: Class features a stable interface and is provided in a library.
    Class is reused by ➜ clients

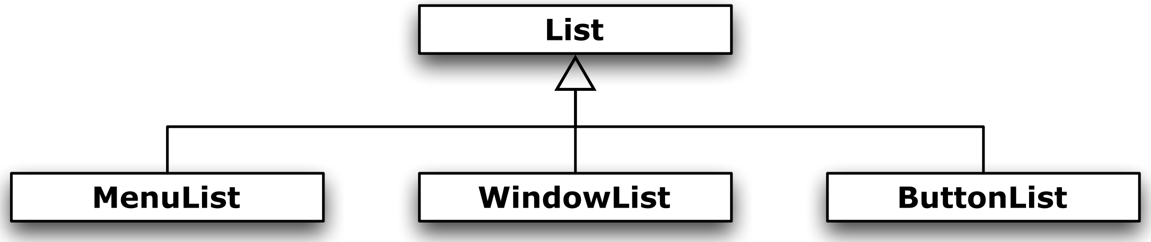
- Two flavors:
  - ① Inheritors have unrestricted access to class implementation
    ➜ Inheritors are affected by changes to the implementation
  - ② Inheritors have no access to class implementation
    ➜ Inheritors are only affected by changes in the interface

- Smalltalk: flavor ①
- Java: fine granularity with access rights

Immutabilty

- How to define an immutable class (in Java)?
  - No mutators or copy on write
    - Copy on write: all mutating operations work on a copy and return it
  - Class must not be extensible
    - Use public final class ...
  - All fields must be final
    - Prohibit changes after initialization
  - All fields must be private
    - Prohibit changes to final fields, e.g. non-zero arrays
  - Exclusive access to mutable components
    - References to mutable objects are not available to other clients

- What if implementations of a class only differ by type?



  - Violates principle of substitutability!

- Implementation based on type “Object” is questionable
  - Frequent type conversions necessary
  - Error prone

```
                 =     A        ();
//
         .   (     B      ());
         .   (              ());
//
B       1 = (B     )           .   (0);
B       2 = (B     )           .   (1); //->
```

Generic Classes

- Class specifies formal type parameters
  - Actual type parameters are provided upon instantiation / inheritance
  - Programming language feature

- Programming languages with generic classes:
  - Eiffel, C++
  - In Java since Java 5

```
            A        < >                < >

                      (        )   ;
                   (        )   ;
```

```
    <B      >            =      A        <B      >();
//
          .   (    B      ());
          .   (            ()); //->         -
//
B       1 =            .    (0);
         2 =            .    (1); //->         -
```

Generic Classes

- Java: generic classes are compatible with non-generic classes
  - *Example*: Collection classes can be instantiated without actual type parameter
  - "Object" is assumed upon missing type parameter

- Base types are not allowed for actual type parameters
  - Hardly restricting with since autoboxing / unboxing

- Java reflection (meta information) was extended as well

- Java also provides generic methods

- Generic class have no impact on the JVM
  - Compiler generates compatible byte code
  - Backward compatibility leads to restrictions

Generic Classes

- Advantages of generic classes
  - Reliability (avoiding down-casts)
  - Readability

```
       =     A        ();
//    .
    <B     >   =     A         <B     >();
```

  - Autoboxing / -unboxing complements nicely

```
//                  /-
    <       >        =     A        <       >();
       .   (           (2));
        =         .   (0).        ();

//               /-
       .   (2);
        =         .   (1);
```

- Disadvantage of generic classes
  - Language complexity (syntax and above all semantics)

Inheritance and Generic Classes

- Inheritance is more powerful than genericity
- Combining inheritance and generics is desirable
  - Flexibility and reliability
  - Even more complex syntax and semantics
  - Is List<String> a subtype of List<Object>? (➜ covariance)
- Restricting genericity
  - Actual type parameter must conform to a specific type

```
                  <                >

                  (         )    ;
               (         )    ;
```

```
      <      >            =              <      >
      <      >            =              <      > //                !!
```

- Design of a class API
  - Flexibility and type safety
  - Readability
    - Expressive and self documenting API
    - Client should not care about typing challenges

```
                A  (       <          >  )
      (            :  )          .    .          ( );
```

hardly usable

```
                A  (       <?>   )
      (            :  )          .    .          ( );
```

flexible

- Case study: API design of a generic class “Stack”

```
< >
();
( );
();
();
```

```
//A ( )
A ( < > )
( : ) ( );
```

Compiler ✔

```
//
< > = < >();
< > = ...;
. A ( );
```

Compiler 🚫

```
//A ( )
A ( <? > )
( : ) ( );
```

✔

Inheritance and Generic Classes

```
//A                                       (             )
                A   (C           < >      )
           (!        ())     .    (    ());
```

Compiler ✔

```
//
     <      >              =           <      >();
C          <      >              = ...;
             .   A  (        );
```

Compiler 🚫

```
//A                                       (                      )
                A   (C           <?          >     )
           (!        ())     .    (    ());
```

✔

Inheritance and Generic Classes

- When to use *extends* resp. *super*?
  - **PECS**: **p**roducer **e**xtends, **c**onsumer **s**uper
  - Stack example: src = producer, dst = consumer

		Parameter produces T instances	
		Yes	No
Parameter consumes T instances	Yes	**Type<T>** (*Invariant* in T)	**Type<? super T>** (*Contravariant* in T)
	No	**Type<? extends T>** (*Covariant* in T)	**Type<?>** (*Independent* of T)

Meta-Information

- Information on properties of objects, classes, and the inheritance hierarchy

- Used for:
  - Basic functionality, such as persisting objects
  - Tools (e.g. debugger)
  - Type checking at run-time
  - Java provides meta-information at run-time (reflection)

```
: #
```

```
//
. C . ( , )
```

- Meta classes
  - Usually provided by pure oo languages
  - A class is an object and hence instance of a class (= meta class)
  - Meta classes provide all meta-information
  - Meta-information can be modified

Java Reflection

- Dynamic class loading
- Purpose:
  - Reducing dependencies
  - Multiple implementations of a type are available *but*
    - Only one is needed at run-time *or*
    - Only one is executable in a specific run-time environment

```
      (              )        ...
     =                               ;
      = C     .        (           ).              ();
```

Java Reflection

- Class “Class” provides access to meta-information at run-time
- Example: calling a private method from outside its declaring class

```
                    (             )        ...
          =     A        ();
    .   (          ());
C           =     .   C    ();
              =      .                 (            ,    .     );
      .   A         (    );
      .      (    , 0);
```

Aspect-Oriented Programming

- Limitations of object-oriented programming
  - Factorization is only possible:
    - along the class hierarchy
    - by delegation
  - Retrospective changes are hardly possible

- Cross-cutting concerns
  - Identical functionality spread across a software system
  - Example: database access
    - Functionality: caching, authorization, transaction, logging
  - Ramifications:
    - Duplicated code
    - Mixing domain logic and technical aspects
    - Decreased maintainability: complex and replicated code
    - Reusability questionable: technical aspects are platform dependent

- Aspect-Orientation is an add-on to rather than a replacement of OO!

Aspect-Oriented Programming

- AOP: dividing a problem into aspects rather than objects/classes

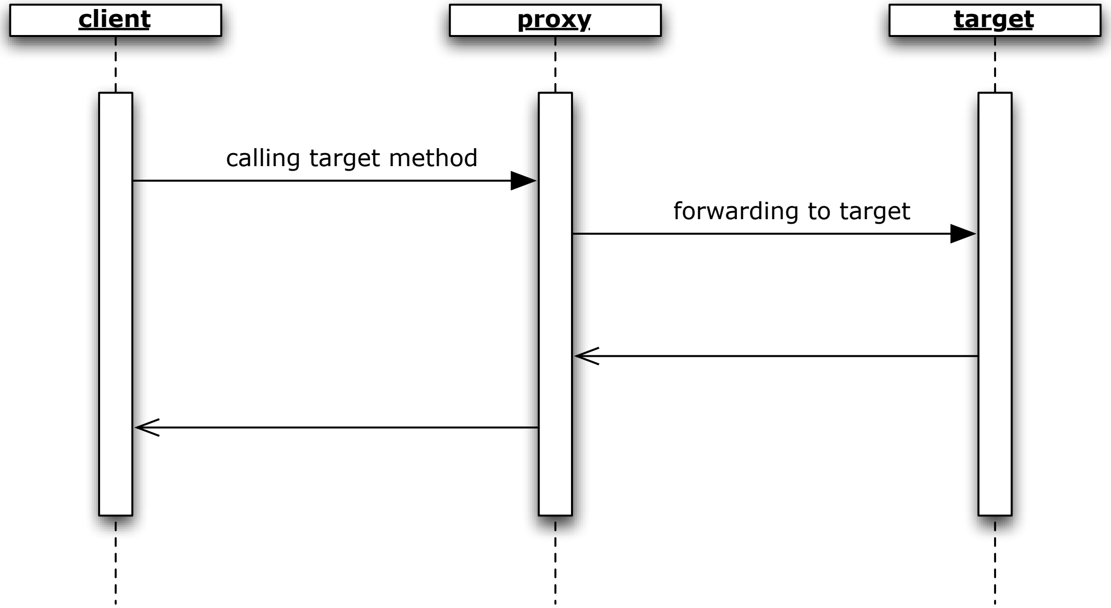
- Join-point model
  - Join Point
    - Point in a running programm where behaviour can be added
    - Examples: method call, access to instance variable etc.
  - Advice:
    - Defines the behaviour (before, after, and around)
  - Point Cut:
    - Quantify join points for executing an advice
    - Point cuts can be defined via pattern matching

- AspectJ: aspects as an extension to Java for AOP
  - Defines point cuts and features advice implementations
  - Can add new instance and class variables / methods
  - Corresponds to a class in object-oriented programming
  - Since Java 5 aspects can be specified using annotations

AOP – Dynamic Proxies

- Application of reflection
- Enables simple aspect-oriented programming

- Task:
  - Adding aspects to classes at run-time
  - No source code changes to these classes
  - No inheritance (base class cannot be changed in retrospect)



AOP – Dynamic Proxies

- Class . . .
  - Facilitates creating classes at run-time
  - Classes may implement several interfaces

AOP – Dynamic Proxies

- Example: logging aspect
  - InvocationHandler logs before and after a method call

```
                                          (          ,             )
                  .                      (   .  C    ().  C          (),
                                              .  C    ().              (),
                                                                       (   ,        ));



                        ;
                        ;


                                         (            ,             )
              =      ;
              =      ;
```

AOP – Dynamic Proxies

```
        (           ,           ,           )

      =    ;
.    ( C          +     .     ());

   =     .    (     ,    );
 (                  )
   .   (    .     () +        + .  C   ());
   .  C   ();

.    (            +     .      ());
     ;
```

- Example of usage

```
= ( ) . ( (),
()); 
```

AOP – Google Guice

- Google Guice (pronounced “juice”)
  - Lightweight dependency injection framework
- Dependency Injection
  - Decoupling and configuration of dependencies
- Google Guice provides some infrastructure for AOP

AOP – Google Guice

- Approach
  - Module
    - Definition of advices
    - Objects must be created by injector
  - Interceptor specifies advice
  - Annotations can be used to define join point

```
@          (                .        ) @        (             .       )
@
```

```
                 =           ();

                      (                            )
              .    ("B           "  +        .        ().      ());
                    =           .       ();
              .    ("           "  +        .        ().      ());
                   ;
```

AOP – Google Guice

```
                  A

                     ()
                ((          .    (),              //                ?
        .                  (          .       ),   //                 ?
                           ());               //


                    (                 )
                  =         .                  (          ());
               =          .                 (     .      );
      .    ();
```

```
@
                       ()
   //...
```

AOP – Google Guice

- Internals
  - Google Guice dynamically creates subclasses
  - Subclass overrides method and invokes interceptor

- Appraisal
  - Google Guice offers simple AOP for common use cases
    - Lightweight solution
    - Simpler but not as powerful as comprehensive AOP approaches, e.g. Spring
    - Way more usable than dynamic proxies (and more abstract, too)
  - Exemplary API
    - Flexibility and expressivity
  - Creating objects via injector can be cumbersome
    - Viral impact
    - Adding AOP retrospectively is hardly possible
      - Even if using patterns rather than annotations for point cuts
  - Debugging
    - Noisy run-time stack

Polymorphism - Java Examples

- Parametric polymorphism ➜ generic classes
- Inheritance ➜ chapter “Advanced Principles” and “Class Libraries”
- Coercion ➜ e.g. autoboxing/-unboxing
- Overloading

```
                    (         )   ...
                    (        )   ...


//...

              =           ();

      .     (256);
      .     (            );
```

Literature

- Concepts
  - H. Züllighoven: Das objektorientierte Konstruktionshandbuch
  - B. Meyer: Object Oriented Software Construction
- Reflection in Java
  - I. Forman, N. Forman: Java Reflection in Action
- Java
  - Java Generics: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
  - J. Bloch: Effective Java
- Values
  - D. Bäumer et al: Values in Object Systems
- Aspect-oriented programming
  - G. Kiczales et al: Aspect-Oriented Programming
  - AspectJ: http://www.eclipse.org/aspectj/
  - Google Guice: http://code.google.com/p/google-guice/