Inheritance

Slide Series 6

Content

Implementation inheritance
Hiding
Inheritance and initialization, canonical form
Variances
Pre and post-conditions
Sub type vs subclass
Liskov substitution principle
Fragile base class problem
Inheritance and immutability
DP: Template and Decorator

Kinds of Inheritance

Interface inheritance

- **implements**
- Java supports multiple interface inheritance

Implementation inheritance

- **extends**
- Java supports single implementation inheritance

Interface Inheritance

No problems

Separates "contract" and implementation, reduces dependencies, basic principle

Implementation Inheritance

A way to reuse code. Sometimes handy but also many problems, .. we'll see quite a few

If A : > B (classes)

- All B's have a sub object of type A
- B objects are a subset of A objects (All B's are A's but not all A's are B's

From now we only talk implementation inheritance

Implementation Inheritance Usage

Remove duplicate code (using abstract superclass)

Extend class with more behavior

- Never exclude from superclass

Some design patterns (Template, ...)

Class Members

Members of a class is given by a class member declaration // JLS 8.1.6

```
// A field (attribute), a method, an inner class,
// etc are members
ClassMemberDeclaration:
    FieldDeclaration
    MethodDeclaration
    ClassDeclaration
    InterfaceDeclaration
    ;
```

What gets Inherited?

Members ... from its direct superclass (§8.1.4) [interface], except in class Object, which has no direct superclass

"A class C inherits from its direct superclass and direct superinterfaces all abstract and non-abstract methods of the superclass and superinterfaces that are public, protected, or declared with default access in the same package as C, and are neither overridden (§8.4.8.1) nor hidden (§8.4.8.2) by a declaration in the class." /JLS 8.4.8.

Constructors, static initializers, and instance initializers are not members and therefore are not inherited.

Protected and Final

Access specification: **protected**

- Visible to subclasses and types in same package
- .. some subtle restrictions, see JLS 6.6.2.

New usage of **final***

- Will prohibit inheritance on class
- Will prohibit overriding for method)

Super*

Keyword **super**

"The form super.Identifier refers to the field named Identifier of the current object, but with the current object viewed as an instance of the superclass of the current class.

The form T.super.Identifier refers to the field named Identifier of the lexically enclosing instance corresponding to T, but with that instance viewed as an instance of the superclass of T.

The forms using the keyword super are valid only in an instance method, instance initializer, or constructor, or in the initializer of an instance variable of a class. If they appear anywhere else, a compile-time error occurs.

These are exactly the same situations in which the keyword this may be used (§15.8.3)." // JLS 15.11.2

Hiding*

"If a class declares a static method m, then the declaration m is said to hide any method m', where the signature of m is a [sub] signature (§8.4.2) of the signature of m', in the superclasses and superinterfaces of the class that would otherwise be accessible to code in the class. It is a compile-time error if a static method hides an instance method."

But: A static attribute can hide an instance attribute

Possible to access hidden

"A hidden method can be accessed by using a qualified name or by using a method invocation expression (§15.12) that contains the keyword super or a cast to a superclass type." //JLS 8.4.8.2

Inheritance and Initialization*

Constructor called last in initialization sequence (static -> instance -> constructor)

If subclass superclass constructor called first in subclass constructor

Assume: A :> B :> C and new C();

Call chain: C ctor -> B ctor -> A ctor -> B ctor -> C ctor

"this"

"When used as a primary expression, the keyword **this** denotes a value that is a reference to the object for which the instance method was invoked (§15.12), or to the object being constructed.

The type of **this** is the class C within which the keyword this occurs.

At runtime, the class of the actual object referred to may be the class C or any subclass of C.

The keyword this is also used in a special explicit constructor invocation statement, which can appear at the beginning of a constructor body (§8.8.7)."
//JLS 15.8.3

"this", cont*

"this" has type as enclosing class but possible hold reference to subclass object

```
class A {
    public A(){
        this...    // Type of reference "this" is A
    }              // but possible referencing B object

class B extends A(){
    // Here type of this is B
}
```

Inheritance and Canonical Form

Have seen the canonical operations (inherited from object)

- hashCode()
- equals()
- clone()

How do inheritance affect... ?

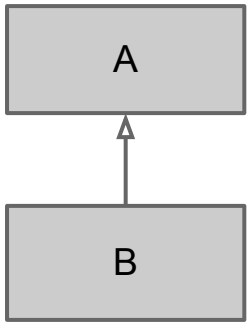
Equals

The equals method implements an equivalence relation:

- It is reflexive: for any reference value x, x.equals(x) should return true.
- It is symmetric: for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is transitive: for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent: for any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
- For any non-null reference value x, x.equals(null) should return false.

If we have inheritance hard to fulfill all this (have to do workaround)!!

Equals: Same or Mixed Type



A a = new A()
B b = new B();

a.equals(b) == ???

Should we allow sub and super classes to be equal or ...?

"[same type equals]...then check to see whether some object of the subclass is equal to an object of the super class, even if the objects are equal in all important aspects, you will get the surprising answer that they aren't equal. In fact, this violates a strict interpretation of the Liskov substitution principle [upcoming], and can lead to very surprising behavior."// Josh Bloch (Java guru)

Equals: Implementation*

Same type equals() has the following lines

```
// Get runtime type
if (getClass() != other.getClass()){
    return false;
}
```

Mixed type equals() has the following lines

```
// In type A.
// Subtype ok! Also handle null
if (!(other instanceof A)) {
    return false;
}
```

Equals: Significant Attributes*

If mixed type which attributes to use in equals()?

- Should we allow attributes from super in sub equals()?
- If comparing sub and super should we skip attributes from sub?
- .. next slide

Equals: Workaround*

Workaround

- Use single id-attribute in whole inheritance hierarchy
- Let equals use id in comparison
- Make equal and hashCode final (so can't override)

Comparable

General Contract

- anticommutation : x.compareTo(y) is the opposite sign of y.compareTo (x)
- exception symmetry : x.compareTo(y) throws exactly the same exceptions as y.compareTo(x)
  transitivity : if x.compareTo(y)>0 and y.compareTo(z)>0, then x. compareTo(z)>0 (and same for less than)
- if x.compareTo(y)==0, then x.compareTo(z) has the same sign as y. compareTo(z)

consistency with equals is highly recommended, but not required : x.compareTo(y)==0, if and only if x.equals(y) ; consistency with equals is required for ensuring sorted collections (such as TreeSet) are well-behaved.

When a class extends a concrete Comparable class and adds a significant field, a correct implementation of compareTo cannot be constructed. The only alternative is to use composition instead of inheritance.

Clone*

No fundamental impact because of inheritance but...

- Only top level class need to handle exception, subclasses no exception handling
- Special behaviour of super explains the idiom super.clone() in clone method (vs using constructor)

Copy Constructors*

Alternative to clone

```
//Usage of copy constructor
A a = new A();
A a2 = new A(a);  // Pass in original
```

Possible to solve all problems with clone

- But the majority och existing code uses/depends on clone … :-(

Variance*

Possible to relax type checking

Covariance

- Subclass method can return subclass type of superclass method returntype (Co meaning; sub -> sub)
- Possible in Java

Contra variance

- Subclass method can take parameters of superclass type of superclass method parameters (Contra meaning sub->super)
- Not in Java, will end up as overload

Inheritance and Exceptions*

Covariant exceptions allowed

See Design slides

Pre and Post Conditions

Used for method specifications

- A precondition is a predicate assumed to hold before the execution of a method starts
- A postcondition is a predicate that is guaranteed to hold after execution of the method (if precondition holds before)

Pre and Post Conditions, cont

```
//  Some method
public ReturnType doIt( ParamType p ){
}
```

Common preconditions

p != null, p > 0, comparable/serializable, ... :> p, p.size() > 0

Common postcondition

result != null, this.contains(p), ...

Pre and Postconditions Example

// java.util.HashSet
public boolean remove(Object o)

[informal specification]
"Removes the specified element from this set if it is present. More formally, removes an element e such that (o==null ? e==null : o.equals(e)), if this set contains such an element. Returns true if this set contained the element (or equivalently, if this set changed as a result of the call). (This set will not contain the element once the call returns.)"

Subclass vs Subtype

Should always be possible to use subtype instead of super

In Java a subclass is a subtype (using **extends** or **implements**, it's purely syntactic)

...but easy to create subclass that can't replace super (even if they have same methods)

- **A subclass that isn't a subtype!!!**

Liskov Substitution Principle*

When is a subclass a subtype?

LSP defines!

"Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T." // Barbara Liskov (famous CS person)

LSP is a semantic definition of subtyping (vs. Java's syntactic)

Liskov Substitution Principle, cont

Subclass must

- Not break any invariants of superclass
- Must not have stronger preconditions (methods)
- Must not have weaker postconditions (methods)
- Covariance for return types, contra for parameters [not in Java], covariance for exceptions

Fragile Base class Problem*

"...fundamental architectural problem of object-oriented programming systems where base classes (superclasses) are considered "fragile" because seemingly safe modifications to a base class, when [implementation] inherited by the derived classes, may cause the derived classes to malfunction." // Wikipedia

There is a possibility of very strong coupling between super and subclasses!

FBC Examples

Base class calling own methods that can be overridden

Possibly introduce overloading or overriding if new methods in baseclass

If subclass directly uses attributes from superclass

Inheritance and Representation Exposure*

Inheritance easily breaks immutability

- Override protected method and change access to public (possible representation exposure)

Design Pattern: Template*

An algorithm is the same for many types, except for one "type specific" step.

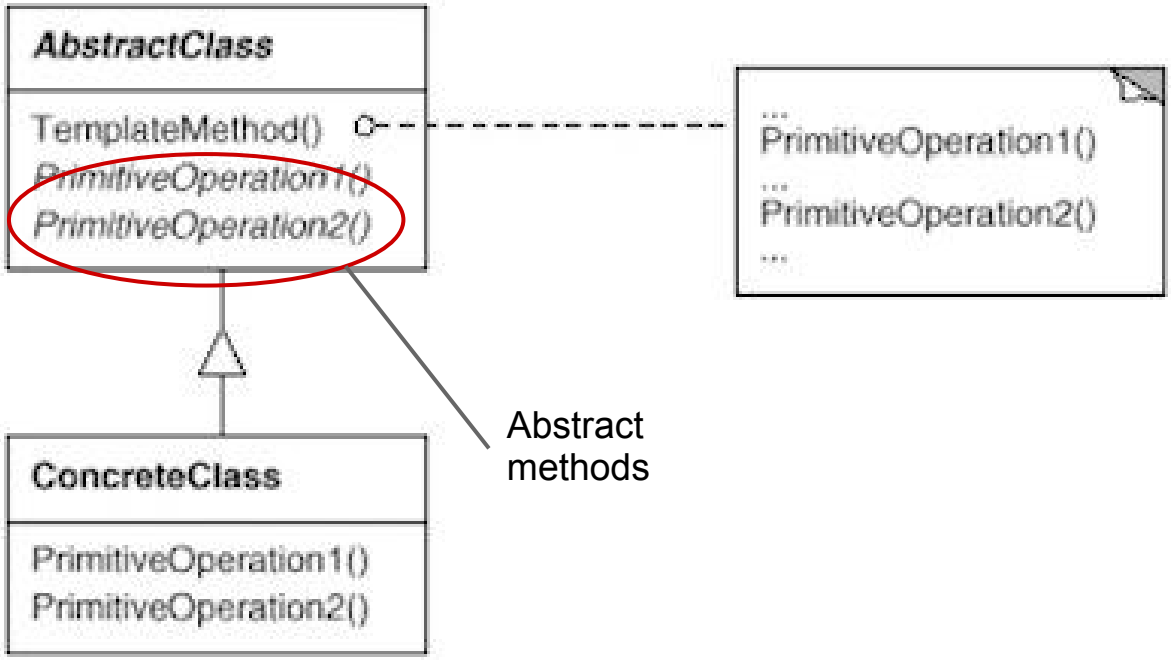
- Put overall algorithm in abstract base class
- Put specific step in subclasses

Benefits

- Eliminate duplicate code
- Use OPC to extend

Need subtyping/override ...

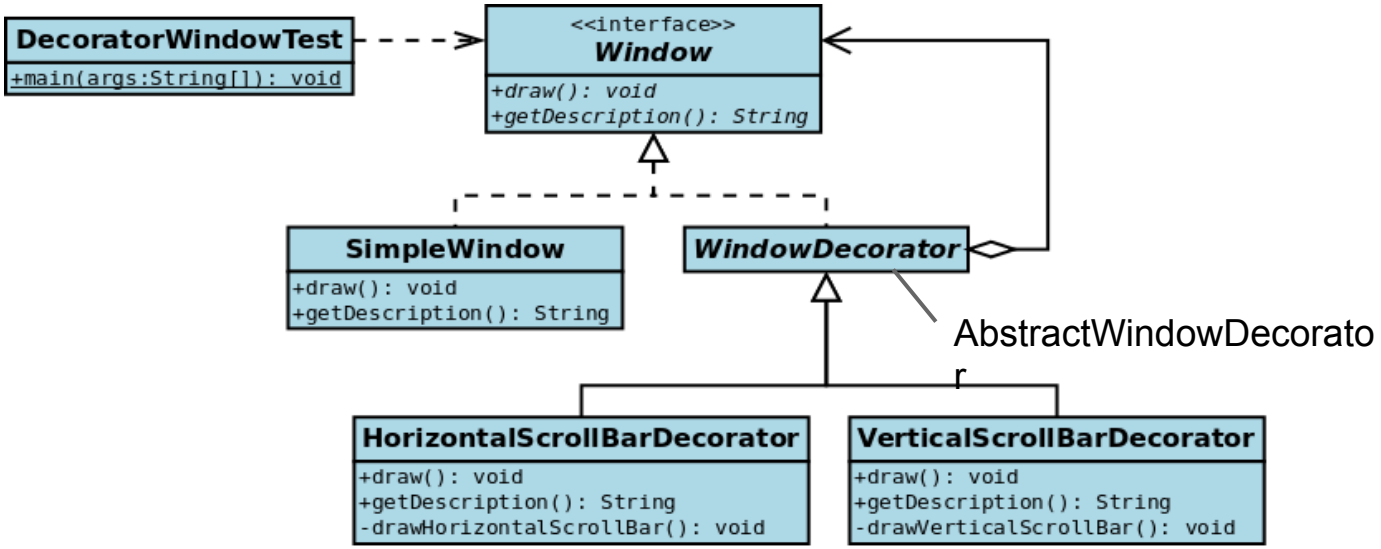
Design Pattern: Template*



Decorator design pattern*

The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at run-time for individual objects (also OPC).

```
Window decoratedWindow = new HorizontalScrollBarDecorator(
                         new VerticalScrollBarDecorator(new SimpleWindow()));
```



Inheritance: Best Practises

- Keep your fields private (and not just package-private).
- Call your own getters/setters.
- Mark classes final, until you have confirmed that they are reliably sub-classable.
- Mark methods final, until you have confirmed that they are reliably overridable.
- Never trust that your class is reliably sub-classable until you have carefully reviewed the design, and written and tested a significant subclass for it.
- Always document clearly how to subclass.

Summary

- Implementation inheritance is very cumbersome, many problems
- Design for inheritance or ban
- Implementation inheritance is static, Decorator as an alternative
- Prefer delegation to implementation inheritance (easier to understand and dynamic)
- Template uses inheritance
- Anyway: Implementation inheritance useful in some situations