Inheritance

Object Oriented Programming

http://softeng.polito.it/courses/09CBI

SOftEng
http://softeng.polito.it

Version 4.14.0

Inheritance

- A class can be a sub-type of another (base) class
- The new (derived) class
  - Implicitly contains (inherits) all the members of the class it inherits from
  - Can augment its structure with any additional member that it defines explicitly
  - Can override the definition of existing methods by providing its own implementation
- The code of the derived class consists of the changes and additions to the base class

Addition

```
class Employee{
   String name;
   double wage;
   void incrementWage(){…}
}
```

```
class Manager extends Employee{
   String managedUnit;
   void changeUnit(){…}
}
```

```
Manager m = new Manager();
m.incrementWage(); // OK, inherited
```

Override

```
class Vector{
   int elements[];
   void add(int x)  {…}
}
```

```
class SortedVector extends Vector{
   void add(int x){…}
}
```


Override

```
class Employee{
 String name;
 public void print(){
   System.out.println(name);
 }
}
```

```
class Manager extends Employee{
  private String managedUnit;
  public void print(){ //override
    System.out.println(name + ”, manages " +
                                managedUnit);
  }
}
```

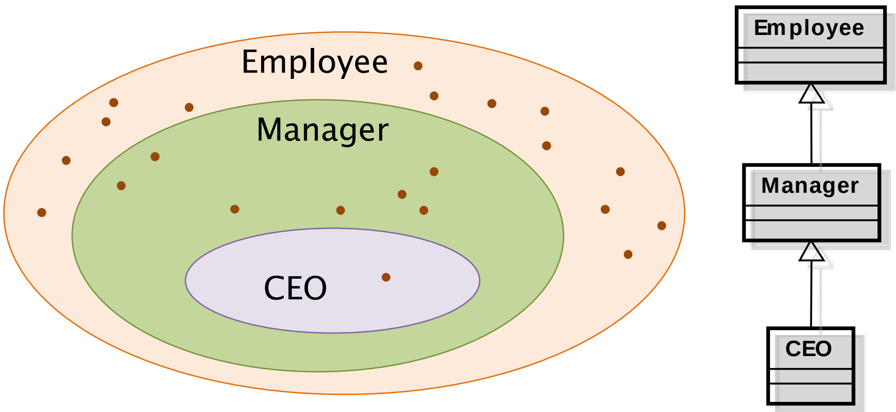
Why inheritance – Reuse

- Often, a class is a minor modification of an already existing class.
- Inheritance avoids code repetitions
- Localization of code:
  - Fixing a bug in the base class automatically fixes it in all the subclasses
  - Adding a new functionality in the base class automatically adds it in the subclasses too
  - Less chances of different (and inconsistent) implementations of the same operation

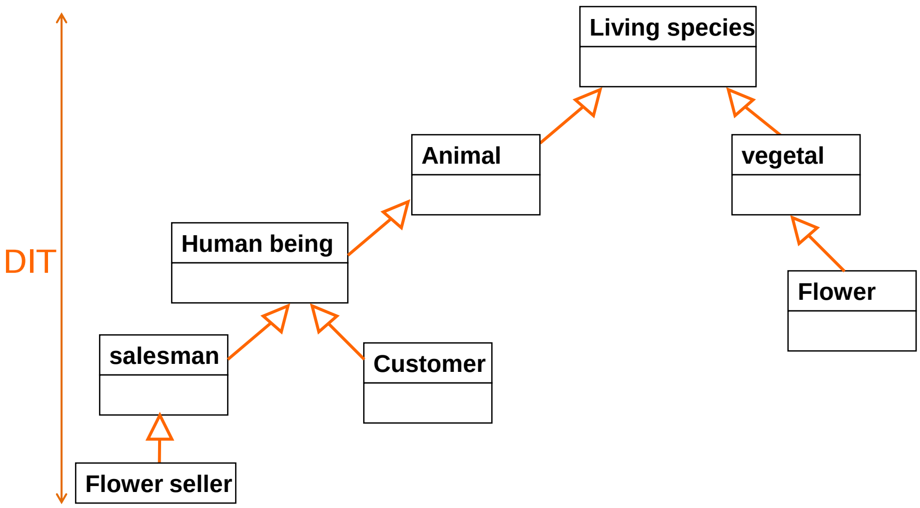
Why inheritance – Flexibility

- Often, we need to treat objects from different classes in a similar way
  - Polymorphism allows feeding algorithms with objects of different classes
    - provided they share a common base class
  - Dynamic binding allows accommodating different behavior behind the same interface

Generalization



Inheritance tree



Depth of Inheritance Tree

- DIT = # levels below root base class
- Too deep inheritance trees reduces code understandability
  - In order to figure out the structure and behavior of a class you need to look into each and every ancestor class
- General rule is to keep DIT $\leq$ 5
  - Empirical limit

Terminology

- Class one above
  - Parent class
- Class one below
  - Child class
- Class one or more above
  - Superclass, Ancestor class, Base class
- Class one or more below
  - Subclass, Descendent class

Polymorphism and Dynamic Binding

Polymorphism

- A reference of type **T** can point to an object of type **S** if-and-only-if
  - **S** is equal to **T** or
  - **S** is a subclass of **T**

```
Employee e;
e = new Employee(); // same class
e = new Manager(); // subclass
```

Polymorphism

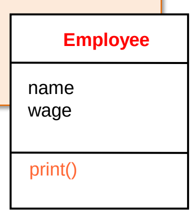
- You can treat indifferently objects of different classes, provided they derive from a common base class

```
Employee[] team = {
  new Manager("Mary Black",25000,"IT"),
  new Employee("John Smith",12000),
  new Employee("Jane Doe",12000)
};
```


Static type checking

- The compiler performs a check on method invocation on the basis of the reference type

```
for(Employee it : team){
  it.print();
}
```



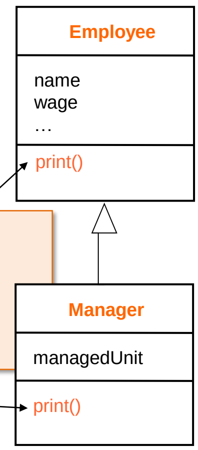
Does the type of `it` (i.e. `Employee`) provide method `print()`?

Dynamic Binding

- Association message - method
  - Performed by JVM at run-time
- Constraint
  - Same signature

```
for(Employee it : team){
   it.print();
}
```

message methods



Dynamic binding procedure

1. The JVM retrieves the effective class of the target object
2. If that class defines the required method, it is executed
3. Otherwise the parent class is considered and step 2 is repeated

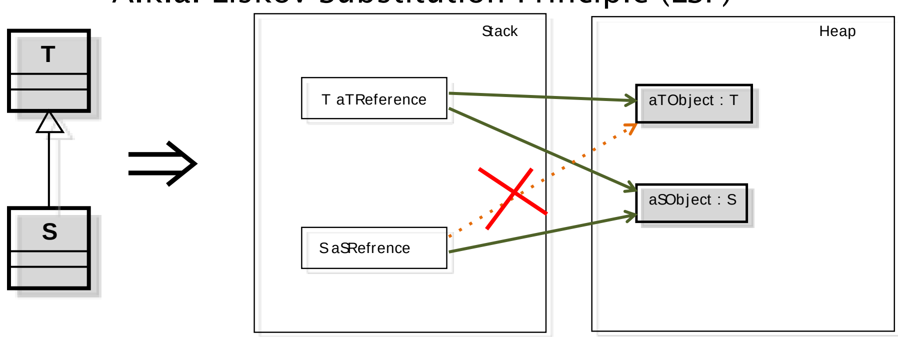
- Note: the procedure is guaranteed to terminate
  - The compiler checks the reference type class (a base of the actual one) defines the method

Why dynamic binding

- Several objects from different classes, sharing a common ancestor class
- Objects can be treated uniformly
- Algorithms can be written for the base class (using the relative methods) and applied to any subclass

Substitutability principle

- If `S` is a subtype of `T`, then objects of type `T` may be substituted with objects of type `S`
  - A.k.a. Liskov Substitution Principle (LSP)



Inheritance vs. Duck typing

- Duck typing
  - Correctness of method invocation is checked at run-time
  - Invocation is correct if the actual class of the target object provides the required method (directly or inherited)
  - Dynamic binding can result into an error

If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck

Override rules

- A method override must use exactly the original method signature
- Visibility cannot be restricted
  - Though it might widen visibility
  - Required to allow a compile-time check on method visibility that is not disrupted at run-time

Override rules

- A slightly different method is not considered as an override by the compiler and therefore not involved in the dynamic binding procedure
  - Minor mistakes in typing might jeopardize correct behavior at run-time
- Using annotation `@Override`
  - Informs the compiler that a method is intended as an override
  - Generates an error if not a correct override

Improper Override

```
class Employee{
  public void print(){ … }
}
```

```
class Manager extends Employee{
  public void Print(){…}
}
```

Unexpected behavior at runtime since this method is no used

```
class Manager extends Employee{
  @Override
  public void Print(){…}
}
```

Error at compile-time this is not an override

CASTING

Types

- Java is a strictly typed language, i.e.,
  - each variable has a type
  - a variable can host only value of that type

```
float f;
f = 4.7;    // legal
f = "string";// illegal
Car c;
c = new Car();  // legal
c = new String(); // illegal
```

Cast – Primitive types

- Type conversion
  - explicit or implicit

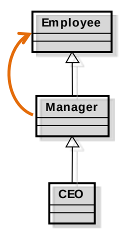
```
int i = 44;
float f = i;
// implicit cast 2c -> fp
f = (float) 44;
// explicit cast
```


Upcast

- Assignment from a more specific type (subtype) to a more general type (supertype)

```
Employee e = new Employee(…);
Manager m = new Manager(…);
Employee em = (Employee) m;
```

  - $\forall$ m $\in$ Manager : m $\in$ Employee



- Upcasts are always type-safe and are performed implicitly by the compiler
  - `Employee em = m;`

Cast and conversion

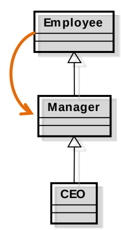
- Reference type and object type are distinct concepts
- A reference cast only affects the reference
  - In the previous example the object referenced to by ‘**`em`**’ continues to be of Manager type
- Notably, in contrast, a primitive type cast involves a value conversion

Downcast

- Assignment from a more general type (supertype) to a more specific one (subtype)

```
Employee em = …;
Manager mm = (Manager)em;
```

  - $\exists$ em $\in$ Employee : em $\in$ Manager
  - $\exists$ em $\in$ Employee : em $\notin$ Manager



- By default it is not safe, no automatic conversion is provided by the compiler
  - Must be explicit
  - Forces the programmer to take responsibility of checking the cast is valid

Downcast

- To access a member defined in a class you need a reference of that class type
  - Or any subclass

```
Employee emp = team[0];
s = emp.getDepartment();
Manager mgr = (Manager) team[0];
s = mgr.getDepartment();
```

Syntax Error: The method `getDepartment()` is undefined for the type `Employee`

Downcast – Warning

- Compiler trusts any downcast
- JVM checks type consistency for all reference assignments, at run-time
  - The class of the object must be equal to the class of the reference or to any of its subclasses

```
mgr = (Manager)team[1];
```

`ClassCastException`: `Employee` cannot be cast to `Manager`

Downcast safety

- Use the **`instanceof`** operator

  `aReference instanceof aClass`

  - Returns true if the object pointed by the reference can be cast to the class
    - i.e. if the object belongs to the given class or to any of its subclasses

```
if(team[1] instanceof Manager){
    mgr = (Manager)team[1];
}
```


Example **`instanceof`**

instanceof → ↓	Employee	Manager	CEO
anEmployee	true	false	false
aManager	true	true	false
aCEO	true	true	true

VISIBILITY (SCOPE)

Example

```
class Employee {
  private String name;
  private double wage;
}
```

```
class Manager extends Employee {

  void print() {
    System.out.println("Manager" +
                               name + " " + wage);
  }
}
```

Not visible

Visibility and inheritance

- Attributes and methods marked as
  - `public` are always accessible
  - `protected` are accessible from within the class, classes in the package and subclasses
  - *package* are accessible from within the class, classes in the package
  - `private` are accessible from within the declaring class only

In summary

	Method in the same class	Method of other class in the same package	Method of subclass	Method of class in other package
`private`	✓			
package	✓	✓		
`protected`	✓	✓	✓	
`public`	✓	✓	✓	✓

Inheritance and constructors

39

39

Construction of child's objects

- Since each object "contains" an instance of the parent class (attributes), the latter must be initialized first
- Java compiler automatically inserts a call to default constructor (w/o parameters) of the parent class
- The call is inserted as the first statement of each child constructor

40

Construction of child objects

- Execution of constructors proceeds top-down in the inheritance hierarchy
- As a consequence, when a method of the child class is executed (constructor included), the super-class is completely initialized already

Example

```
class Employee {
 Employee() {
    System.out.println("ctor Employee"); }
}
```

```
class Manager extends Employee  {
 Manager() {
    System.out.println("ctor Manager"); }
}
```

```
class CEO extends Manager {
 CEO() {
    System.out.println("ctor CEO"); }
}
```

```
new CEO()
```

```
ctor Employee
ctor Manager
ctor CEO
```

Explicit constructors

- If any constructor is explicitly defined the default constructor "disappears"
  - Derived class constructor implicitly invokes the base class constructor that cannot be resolved

```
class Employee{
  …
  Employee(String name, double wage){}
  // no default ctor is defined
}
```

```
class Manager extends Employee{
}
```

Error: Implicit constructor is undefined

Explicit constructors

- An explicit constructor withou arguments can be defined

```
class Employee{
  …
  Employee(String name, double wage){}
  Employee(){} // explicit constructor
}
```

```
class Manager extends Employee{
}
```

super (constructor)

- The child class constructor must call the right constructor of the parent class, explicitly
- Use `super()` to invoke the constructor of parent class and pass the appropriate arguments
- Must be the first statement in child constructors

super (constructor) Example

```
class Employee {
  private String name;
  private double wage;
      ???
  Employee(String n, double w){
    name = n;
    wage = w;
  }
}
```

```
class Manager extends Employee {
  private int unit;

  Manager(String n, double w, int u) {
    super(); // ERROR !!!
    unit = u;
  }
}
```

The compiler adds this implicit invocation

super (constructor) Example

```
class Employee {
  private String name;
  private double wage;

  Employee(String n, double w){
    name = n;
    wage = w;
  }
}
```

```
class Manager extends Employee {
  private int unit;

  Manager(String n, double w, int u) {
    super(n,w);
    unit = u;
  }
}
```


Masked overridden methods

- When a method in a derived class overrides one in the base class, the latter is masked
  - I.e. the overridden method is invisible from the derived class
- This rule might represent a problem if we wish to re-use the original overridden method from within the subclass

super (reference) example

```
class Employee{
 String name;
 public void print(){
   System.out.println(name);
 }
}
```

```
class Manager extends Employee{
  private String managedUnit;
  public void print(){ //override
     super.print();
     System.out.println("\tmanages " +
                               managedUnit);
  }
}
```


super (reference)

- **this** references the current object
- **super** references the parent class

Attributes redefinition

```
class Parent{
   protected int attr = 7;
}
```

```
class Child{
   protected String attr = "hello";

   void print(){
     System.out.println(super.attr);
     System.out.println(attr);
   }
}
```


Final method

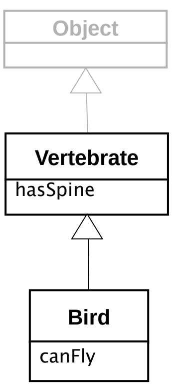
- The keyword **`final`** applied to a method makes it not overridable by subclasses
  - When methods must keep a predefined behavior
  - E.g. method provide basic service to other methods

Not an antinomy:
in Java there is a class called “`Object`”

CLASS `Object`

Class `Object`

- `java.lang.Object`
- All classes are subtypes of Object



extends Object — Implicitly

```
class Vertebrate {
   …
}
```

```
class Bird extends Vertebrate{
   …
}
```

Why class **`Object`**

Object
toString() : String equals(Object) : boolean

- Generality:
  - any instance of any class can be seen as an **`Object`** instance
  - **`Object`** is the universal reference type
- Common behavior:
  - **`Object`** defines some common operations which are inherited by all classes
  - Often, they are overridden in sub-classes

Generality

- Each class is either directly or indirectly a subclass of **`Object`**
- It is always possible to *upcast* any instance to **`Object`** type

```
AnyClass foo = new AnyClass();
Object obj;
obj = foo;
```

- References of type **`Object`** play a role similar to **`void*`** in C

Generality – group of objects

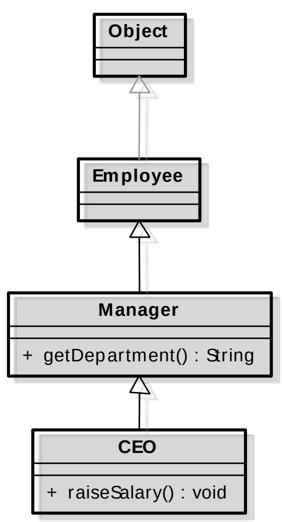
- We can collect heterogeneous objects into a single container

```
Object[] objects = new Object[4];
objects[0]= "First!";
objects[2]= new Employee();
objects[1]= new Integer(2);
Objects[3]= 42;
for(Object obj : objects){
  System.out.println(obj);
}
```

Note: wrappers must be used instead of primitive types

Or leverage autoboxing

Example Company Employees



`Object` class methods

- `toString()`
  - Returns string representation of the object
- `equals()`
  - Checks if two objects have same contents
- `hashCode()`
  - Returns a unique code
- Protected:
  - `clone()`
    - Creates a copy of the current object
  - `finalize()`
    - Invoked by GC upon memory reclamation

`Object.toString()`

- Returns a string representing the object contents
- The default (Object) implementation returns:

  `ClassName@#hashcode#`

  - Es:

  `org.Employee@af9e22`

Object
toString() : String equals(Object) : boolean hashCode() : int

toString() example

```
class Employee {
// …
   @Override
   public String toString(){
      return "Employee: " + name;
   }
}
```


Object.equals()

- Tests equality of values
- Default implementation compares references:

Object
toString() : String equals(Object) : boolean hashCode() : int

```
  public boolean equals(Object other){
     return this == other;
  }
```

- Must be overridden to compare contents

The `equals()` contract

- Reflexive: `x.equals(x) == true`
- Symmetric: `x.equals(y)==y.equals(x)`
- Transitive: for any reference x, y and z
  - if `x.equals(y)==true &&` `y.equals(z)==true` => `x.equals(z)==true`
- Consistent: multiple calls `x.equals(y)` consistently return true (or false)
  - No information used in the comparison should be changed (immutables)
- Robust: `x.equals(null) == false`

`equals()` example

```
@Override
public boolean equals(Object o){
  if( !(o instanceof Employee)){
        return false;
  }
   Employee other = (Employee)o;
   return this.name.equals(other.name);
}
```

Note:
`null instanceof X → false`

`Object.hashCode()`

Object
toString() : String equals(Object) : boolean hashCode() : int

- Returns a (fairly) unique code for the object
- Can be used as
  - index in hash tables
  - quick approximation for equality
- The default implementation (Object) converts the internal address of the object into an integer
- Must be overridden to return codes depending on the contents

The `hashCode()` contract

- Consistent: **`hashCode()`** must return the same value, if no information used in **`equals()`** is modified.
- Equal compliant: if two objects are equal for **`equals()`** method, then calling **`hashCode()`** on the two objects must return the same value
- If two objects are unequal for **`equals()`** method, then calling **`hashCode()`** on the two objects *may* return distinct values
  - producing distinct values for unequal objects may improve the performance of hash tables

`hashCode()` vs. `equals()`

Condition	Required	Not Required (but allowed)
`x.equals(y) == true`	`x.hashCode()== y.hashCode()`	
`x.hashCode() == y.hashCode()`		`x.equals(y)== true`
`x.equals(y) == false`		-
`x.hashCode() != y.hashCode()`	`x.equals(y)== false`	

`System.out.print( Object )`

- `print()` methods implicitly invoke `toString()` on all object parameters

```
class Employee{ String toString(){…} }
 Employee e = new Employee();
System.out.print(e);   // same as...
System.out.print(e.toString());
```

- Polymorphism applies when `toString()` is overridden

```
Object ob = e;
System.out.print(ob); //Employee's toString()
```

Variable arguments – example

```
static void plst(String pre, Object...args){
  System.out.print(pre + " ");
  for(Object o : args){
    if(o!=args[0]) System.out.print(", ");
      System.out.print(o);
  }
  System.out.println();
}
public static void main(String[] args) {
  plst("List:", "A", 'b', 123, "hi!");
}
```

List: A, b, 123, hi!

`new Integer(123)`

`new Character('b')`

ABSTRACT CLASSES

Abstract class

- Often, a superclass is used to define common behavior for children classes
- In this case some methods may have no obvious meaningful implementation in the superclass
- Abstract classes leave the behavior partially unspecified
  - The abstract class cannot be instantiated

Abstract modifier

```
public abstract class Shape {

   private int color;

   public void setColor(int color){
      this.color = color;
   }

   // to be implemented in child classes
   public abstract void draw();
}
```

No method body

Better than:
`System.err.println("Sorry, don't know how to draw this shape");`

Abstract modifier

```
public class Circle extends Shape {

   public void draw() {
      // body goes here
   }
}


Object a = new Shape(); // Illegal: abstract
Object a = new Circle();// OK: concrete
```


Abstract modifier

- The **abstract** modifier marks the method as non-complete / undefined
- The modifier must be applied to all incomplete method **and** to the class

```
public abstract class Shape {
   public void setColor(int color){…}

   // implemented in child classes
   protected abstract void draw();
}
```

No method body

Abstract classes

- A class must be declared **`abstract`** if any of its methods is **`abstract`**
- A class that extends an **`abstract`** class should implement (i.e. override) all the base class **`abstract`** methods
- If any **`abstract`** method is not implemented, then the class must be declared **`abstract`** itself

Example: **`Sorter`**

```
public class Sorter {
  public void sort(Object v[]){
    for(int i=1; i<v.length; ++i)
      for(int j=0; j<v.length-i; ++j){
        if(compare(v[j],v[j+1])>0){
          Object o=v[j];
          v[j]=v[j+1]; v[j+1]=o;
  } } }
  protected int compare(Object a, Object b){
    System.err.println("Someone forgot about"+
                       "the compare() method!");
    return 0; // why not 42?
  }
}
```

What else could we do here?

Example: Sorter

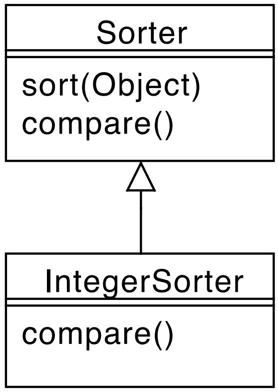
```
public abstract class Sorter {
  public void sort(Object v[]){
    for(int i=1; i<v.length; ++i)
      for(int j=0; j<v.length-i; ++j){
        if(compare(v[j],v[j+1])>0){
          Object o=v[j];
          v[j]=v[j+1]; v[j+1]=o;
    } } }
  }
  abstract int compare(Object a, Object b);
}
```


Example: StringSorter

```
class StringSorter extends Sorter {
  int compare(Object a, Object b){
    String sa=(String)a;
    String sb=(String)b;
    return sa.compareTo(sb);
  }
}
```

```
Sorter ssrt = new StringSorter();
String[] v={"g","t","h","n","j","k"};
ssrt.sort(v);
```

Template Method Example

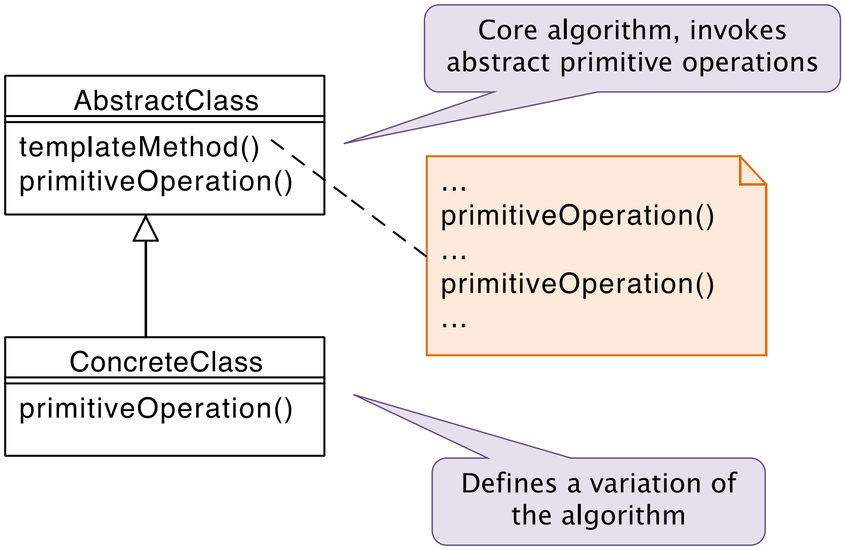


Template Method Pattern

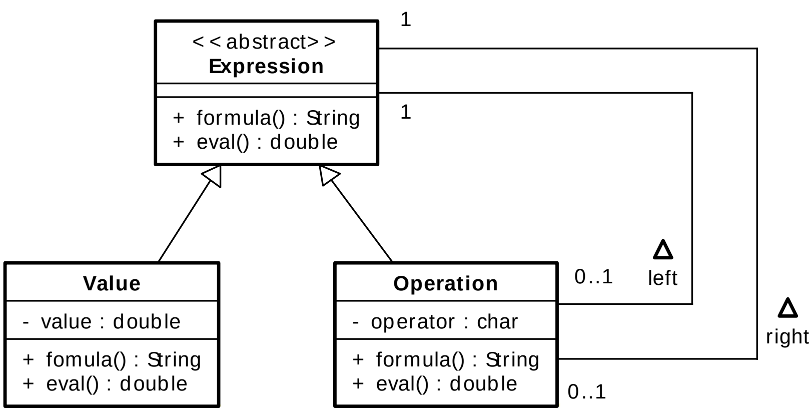
- Context:
  - An algorithm/behavior has a stable core and several variation at given points
- Problem
  - You have to implement/maintain several almost identical pieces of code

See slide deck on design patterns

Template Method



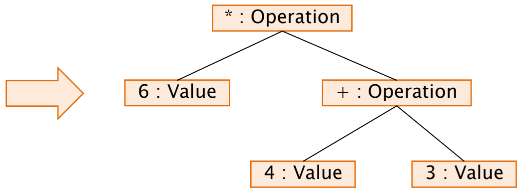
Abstract Expression Tree



Expression Tree example

```
Expression e =
  new Operation('*',
          new Value(6),
          new Operation('+',
                  new Value(4),
                  new Value(3)));
System.out.println( e.formula() + " = "
                   + e.eval());
```

6 * (4 + 3)



Expression Tree example

```
public abstract class Expression{
  public abstract double eval();
  public abstract String formula();
}
```

```
public class Value extends Expression {
  private double value;
  public Value(double v){ value = v; }
  @Override
  public double eval() { return value;
  }
  @Override
  public String formula() {
    return String.valueOf(value);
  }}
```

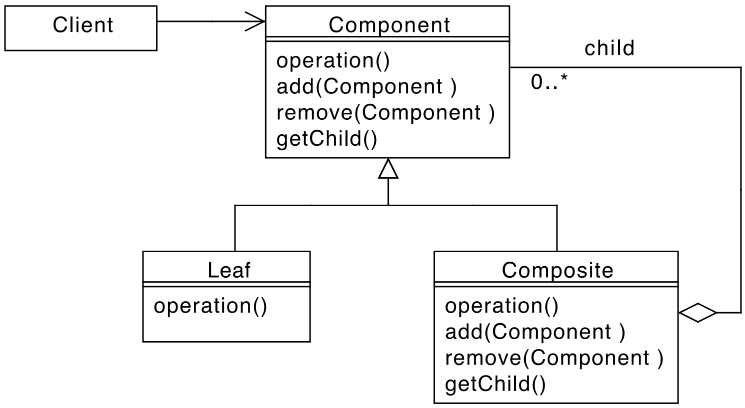
Expression Tree example

```
public class Operation extends Expression {
  private char op;
  private Expression left, right;
  public Addition(char o, Expression l,
                          Expression r){
     op = o; left=l; right=r;
  }
  public double eval() {
   switch(op){
    case '+': return left.eval()+right.eval();
   … }
  }
  public String formula() {
    return "(" + left.formula() + " " + op +
           " " + right.formula() + ")";
} }
```


Composite Pattern

- Context:
  - You need to represent part-whole hierarchies of objects
- Problem
  - Clients need to access a unique interface
  - There are structural difference between composite objects and individual objects.

Composite Pattern



INTERFACES

Java interface

- Special type of class where
  - Methods are implicitly abstract (no body)
  - Attributes are implicitly static and final
  - Members are implicitly public
- Defined with keyword **`interface`**
  - Instead of **`class`**
- Cannot be instantiated
  - i.e. no **`new`** (like abstract classes)
- Can be used as a type for references
  - Like abstract class

Interface example

- All methods are implicitly
  - **`abstract`**
  - **`public`**

```
public interface Expression{
  double eval();
  String formula();
}
```

Interface implementation

- A class **implements** interfaces
- A class implementing an interface must override all interface methods
  - Unless the class is declared abstract

```
class Value implements Expression {
  @Override void value(){ … }
  @Override void formula(){ … }
}
```


91

Interfaces and inheritance

An interface

- cannot extend a class

```
interface Bar extends String {
  void print();
}
```

class

- can extend many interfaces

```
interface Bar extends Orderable, Comparable{
  ...
}
```

interfaces

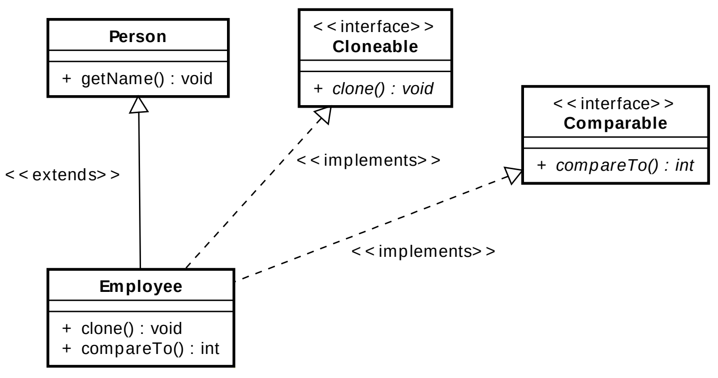
92

Class inheritance

A class can
- extend a single class
  - Single inheritance
- implement multiple interfaces
  - Multiple inheritance

```
class Person extends Employee
implements Cloneable, Comparable
{…}
```


Class inheritance (UML)




```
class Employee extends Person
implements Cloneable, Comparable
{…}
```

Inheritance Classes & Interfaces

	Class ↑	Interface ↑
Class →	**extends** (exactly one)	**implements** (up to many)
Interface →	X	**extends** (up to many)

Anonymous Classes

- Interfaces can be used to instantiate anonymous local classes
  - Within a method code
  - Providing implementation of methods
    - E.g.

```
Iface obj = new Iface(){
  public void method(){…}
};
```

Warning

- In object-oriented jargon the general term *interface* is used to indicate the set of publicly available methods
  - Or a subset, when talking about many *interfaces*
- Java **`interface`**, is a distinct though related concept
  - When a **`class`** implements an **`interface`** the methods defined in the **`interface`** constitute an *interface* of the **`class`**

Static methods in interface

- Interfaces can host **`static`** methods
  - Cannot refer to instance methods
    - Like in regular classes
  - Cannot change static attributes
    - Since they are **`final`** by default in interfaces
  - Can refer to their arguments
  - Can be overridden

    Since Java 8

Default methods

- Interface method implementation can be provided for **`default`** methods
  - Can refer to arguments and other methods
  - Cannot refer to non-static attributes
    - Since they are unknown to the interface
  - Can be overridden in implementing classes as any regular method

Available since Java 8

Default methods: why

- Inject behavior inside interfaces that would otherwise be pure declarations
  - like regular methods in classes, but
  - unlike classes, leverage multiple inheritance
- Pre-existing interfaces can be enhanced – adding new functionality – still ensuring compatibility with existing code written for older versions of those interfaces.

Functional interface

- An interface containing only one regular method
  - May include any additional **`static`** or **`default`** methods
- Method's semantics is purely functional
  - The result of the method is based solely on the arguments
    - i.e. there are no side-effects on attributes
  - E.g., **`java.lang.Comparator`**

Functional interface

- Predefined interfaces are defined in
  - **`java.util.function`**
  - Specific for different primitive types
  - Generic version (see Generics)
- The predefined interfaces can be used to define behavioral parameterization arguments
  - E.g. strategy objects

Functions

- `Consumer`
  - `void accept(Object value)`
- `Supplier`
  - `Object get()`
- `Predicate`
  - `boolean test(Object value)`
- `Function`
  - `Object apply(Object value)`
- `BiFunction`
  - `Object apply(Object l, Object r)`

Simplified versions of the interfaces in `java.util.function`: actual ones use Generics

Functions (`int` versions)

- `IntFunction`
  - `Object apply(int value)`
- `IntConsumer`
  - `void accept(int value)`
- `IntPredicate`
  - `boolean test(int value)`
- `IntSupplier`
  - `int getAsInt()`
- `IntBinaryOperator`
  - `int applyAsInt(int left, int right)`

INTERFACE USAGE PATTERNS

Purpose of interfaces

- Define a common *"interface"*
  - Allows alternative implementations
- Provide a common behavior
  - Define method(s) to be called by algorithms
- Enable behavioral parameterization
  - Encapsulate behavior in an object parameter
- Enable communication decoupling
  - Define a set of callback method(s)
- Allow class flagging

Alternative implementations

- Context
  - The same module can be implemented in different ways by distinct classes with varying:
    - Storage type or strategy
    - Processing
- Problem
  - The classes should be interchangeable
- Solution
  - An interface defines methods with a well-defined semantics and functional specification
  - Distinct classes can implement it

Alternative implementations

- Complex numbers

```
public interface Complex {
  double real();
  double im();
  double mod();
  double arg();
}
```

  - Can be implemented using, e.g., either cartesian or polar coordinates

Alternative implementations

```
class ComplexRect implements Complex {
  private double im, re;
  public ComplexRect(double re, double im) {
              this.im = im; this.re = re; }
  @Override public double real() { return re; }
  @Override public double im() { return im; }
  @Override public double arg() {
              return Math.atan2(im, re); }
  @Override public double mod() {
              return Math.sqrt(re*re+im*im); }
}
```


Alternative implementations

```
class ComplexPolar implements Complex {
   private double mod, arg;
   public ComplexPolar(double mod, double arg) {
             this.mod = mod; this.arg = arg; }
   @Override public double real() {
              return mod*cos(arg); }
   @Override public double im() {
              return mod*sin(arg); }
   @Override public double mod() { return mod; }
   @Override public double arg() { return arg; }
}
```

Alternative Implementations

- Sample usage

```
Complex c1 = new ComplexRect(4,3);
System.out.println(c1 +
" -> Module " + c1.mod() +
"  argument: " + c1.arg());
Complex c2 = new ComplexPolar(5,0.6435);
System.out.println(c2 +
" -> Real " + c1.real() +
"  Imaginary: " + c1.im());
```


Static methods in interface

- Interfaces can become the façade for alternative implementations
  - If alternatives are known in advance, static methods can serve as factory methods

Interface static methods: ex.

Definition

```
public interface Complex {
  // …
  static Complex fromRect(double re,
                          double im){
    return new ComplexRect(re,im);
  }
  static Complex fromPolar(double mod,
                         double arg){
    return new ComplexPolar(mod,arg);
} }
```


Interface static methods: ex.

Sample usage

```
Complex c1 = Complex.fromRect(4,3);
System.out.println(c1 +
          " -> Module " + c1.mod() +
          "  Argument: " + c1.arg());
Complex c2 = Complex.fromPolar(5,0.6435);
System.out.println(c2 +
          " -> Real " + c2.real() +
          "  Imaginary: " + c2.im());
```

Default methods: example

```
public interface Complex {
  double real();
  double im();
  double mod ();
  double arg();
  default boolean isReal(){
    return im()==0;
  }
}
```


Common behavior

- Context
  - An algorithm requires its data to provide a predefined set of common operations
- Problem
  - The algorithm must work on diverse classes
- Solution
  - An interface defines the required methods
  - Classes implement the interface and provide methods that are used by the algorithm

Common behavior: sorting

- Class **`java.utils.Arrays`** provides the static method **`sort()`**

```
int[] v = {7, 2, 5, 1, 8, 5};
Arrays.sort(v);
```

- Sorting object arrays requires a means to compare two objects:
  - **`java.lang.Comparable`**

Comparable

– Interface **`java.lang.Comparable`**

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

- Returns
  - <0 if **`this`** precedes **`obj`**
  - =0 if **`this`** equals **`obj`**
  - >0 if **`this`** follows **`obj`**

Note: simplified version, actual declaration uses generics

Example of **`Comparable`** usage

```
public class Student
        implements Comparable {
  int id;
  Student(int id){ this.id=id; }
  @Override
  public int compareTo(Object o){
    Student other = (Student)o;
    return this.id - other.id;
  }
}
```


Common behavior: iteration

– Interface **`java.lang.Iterable`**

```
public interface Iterable {
  Iterator iterator();
}
```

- Any class implementing **`Iterable`** can be the target of a *for-each* construct
  - Uses the Iterator interface

Note: simplified version, actual declaration uses generics

Common behavior: iteration

- Interface `java.util.Iterator`

```
public interface Iterator {
  boolean hasNext();
  Object next();
}
```

- Semantics:
  - Initially positioned before the first element
  - **`hasNext()`** tells if a next element is present
  - **`next()`** returns the next element and advances by one position

Note: simplified version, actual declaration uses generics

Using an iterator

- Iterator loop

```
Iterator it = seq.iterator()
while( it.hasNext() ){
  Object element = it.next();
  System.out.println(el);
}
```

- Equivalent to

```
for(Object element : seq){
  System.out.println(el);
}
```

for-each

`Iterable` example

```
public class Letters implements Iterable {
  private char[] chars;
  public Letters(String s){
     chars = s.toCharArray(); }
  public Iterator iterator(){ return new LI();}
  class LI implements Iterator {
    private int i=0;
    public boolean hasNext(){
      return i < chars.length;
    }
    public Object next() {
      return new Character(chars[i++]);
  } }
}
```

Inner class

`Iterable` example

```
public class Letters implements Iterable {
  private char[] chars;
  public Letters(String s){
     chars = s.toCharArray(); }
  public Iterator iterator() {
    return new Iterator(){
      private int i=0;
      public boolean hasNext(){
        return i < chars.length;
      }
      public Object next() {
        return new Character(chars[i++]);
      }
    };
} }
```

Anonymous inner class

Iterable example

- Usage of an iterator with for-each

```
Letters l = new Letters("Sequence");
for(Object e : l){
  char c = ((Character)e);
  System.out.println(c);
}
```


Iterable example

```
class Random implements Iterable {
  private int[] values;
  public Random(int n, int min, int max){ … }
  class RIterator implements Iterator {
    private int next=0;
    public boolean hasNext() {
         return next < values.length; }
    public Object next() {
         return new Integer(values[next++]);}
  }
  public Iterator iterator() {
    return new RIterator();
  }
}
```

`Iterable` example

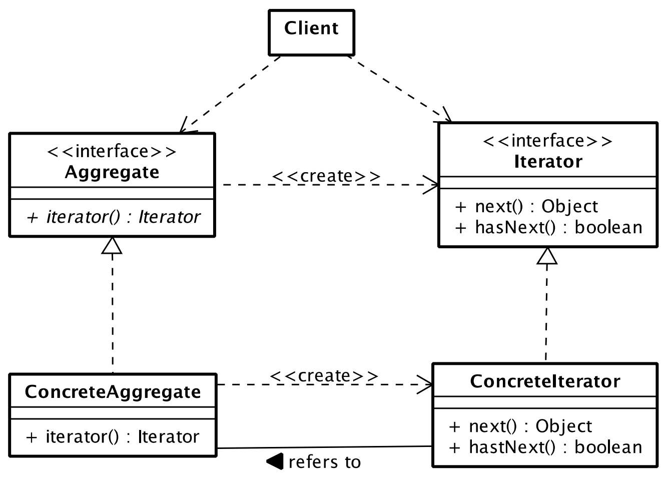
- Usage of an iterator with for-each

```
Random seq = new Random(10,5,10);
for(Object e : seq){
  int v = ((Integer)e).intValue();
  System.out.println(v);
}
```


Iterator pattern

- Context
  - A collection of objects must be iterated
- Problem
  - Multiple concurrent iterations are possible
  - The internal storage must not be exposed
- Solution
  - Provide an iterator object, attached to the collection, that can be advanced independently

Iterator pattern



Behavioral parameterization

- Context
  - A generic algorithm is fully defined but a few given core operations that vary often
- Problem
  - Multiple implementations with significant code repetitions
  - Complex conditional structures
- Solution
  - The operations are defined in interfaces
  - Objects implementing the interface are used to parameterize the algorithm

Behavioral parameters

```
Static void process(Object[] v, Consumer c){
  for(Object o : v){
    c.accept(o);
  }
}
```

```
public interface Consumer{
  void accept(Object o);
}
```

```
String[] v = {"A", "B", "C", "D"};
Processor printer = new Printer();
process(v, printer);
```

```
public class Printer
implements Consumer{
  public void accept(Object o){
    System.out.println(o);
}}
```


Behavioral parameters

```
void process(Object[] v, Consumer c){
  for(Object o : v){
    c.accept(o);
  }
}
```

```
public interface Consumer{
  void accept(Object o);
}
```

```
String[] v = {"A", "B", "C", "D"};
Processor printer = new Consumer(){
  public void accept(Object o){
    System.out.println(o);
  }};
process(v,printer);
```

Anonymous inner class

Behavioral objects

- Objects of classes with no attributes that implement interfaces
- The sole purpose of behavioral objects is being used within algorithms in order to parameterize the behavioral
- Often behavioral objects implement functional interfaces

Strategy Pattern

- Context
  - Many classes or algorithm has a stable core and several behavioural variations
    - The detailed operation performed may vary
- Problem
  - Several different implementations for the variations are needed
  - Usage of multiple conditional constructs would tangle up the code

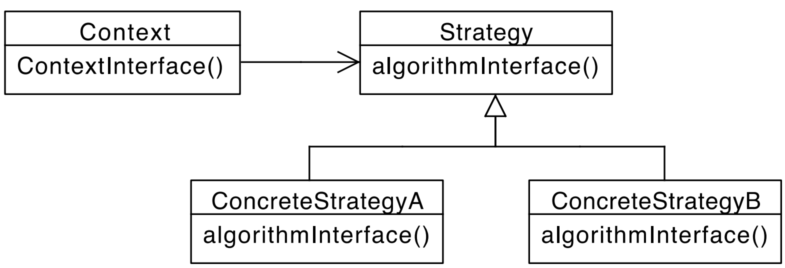
Strategy Pattern

- Solution
  - Embed each variation inside a strategy object passed as a parameter to the algorithm
  - The strategy object's class implements an interface providing the operations required by the algorithm

135

135

Strategy Pattern



136

136

Comparator

– Interface `java.util.Comparator`

```
public interface Comparator{
  int compare(Object a, Object b);
}
```

- Semantics (as comparable): returns
  - a negative integer if **a** precedes **b**
  - 0, if **a** equals **b**
  - a positive integer if **a** succeeds **b**

Note: simplified version, actual declaration uses generics

Comparator

```
public class StudentCmp implements Comparator{
  public int compare(Object a, Object b){
    Student sa = (Student)a;
    Student sb = (Student)b;
    return a.id - b.id;
  }
}
```

```
Student[] sv = {new Student(11),
                new Student(3),
                new Student(7)};
Arrays.sort(sv, new StudentCmp());
```

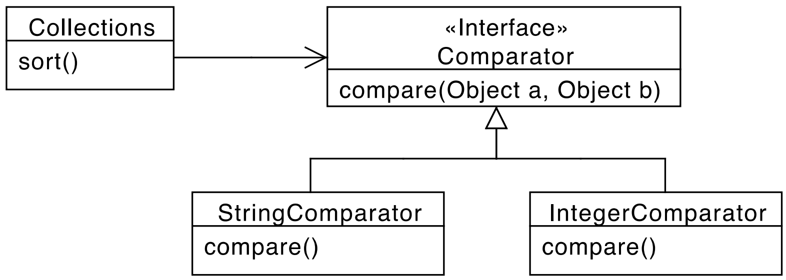
Comparator

```
Student[] sv = {new Student(11),
                 new Student(3),
                 new Student(7)};
Arrays.sort(sv, new Comparator(){
  public int compare(Object a, Object b){
    Student sa = (Student)a;
    Student sb = (Student)b;
    return a.id - b.id;
  }
});
```


Comparator (descending)

```
Student[] sv = {new Student(11),
                 new Student(3),
                 new Student(7)};
Arrays.sort(sv, new Comparator(){
  public int compare(Object a, Object b){
    Student sa = (Student)a;
    Student sb = (Student)b;
    return -(a.id - b.id);
  }
});
```

Strategy Example: `Comparator`



Strategy Consequences

+ Avoid conditional statements
+ Algorithms may be organized in families
+ Choice of implementations
+ Run-time binding

– Clients must be aware of different strategies
– Communication overhead
– Increased number of objects

Communication decoupling

- Separating senders and receivers is important to:
  - Reduce code coupling
  - Improve reusability
  - Enforce layering and structure

Observer - Observable

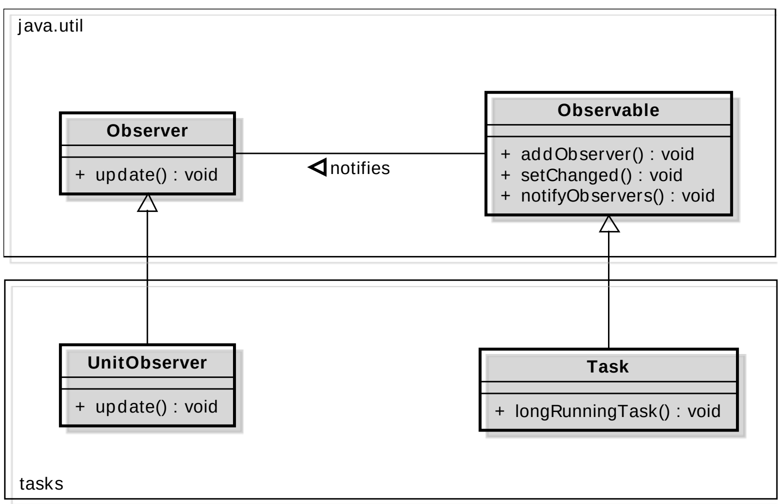
- Allows a standardized interaction between an objects that needs to notify one or more other objects
- Defined in package **`java.util`**
- Class **`Observable`**
- Interface **`Observer`**

Warning: Observer-Observable have been deprecated in Java 9

Observer - Observable

- Class **`Observable`** manages:
  - registration of interested observers by means of method **`addObserver()`**
  - sending the notification of the status change to the observer(s) together with additional information concerning the status (event object).
- Interface **`Observer`** allows:
  - Receiving standardized notification of the observer change of state through method **`update()`** that accepts two arguments:
    - Observable object that originated the notification
    - Additional information (the event object)

Observer - Observable



Observer - Observable

- Sending a notification from an observable element involves two steps:
  - record the fact the the status of the observable has changed, by means of method **`setChanged()`**,
  - send the actual notification and provide additional information (the event object), by means of method **`notifyObservers()`**

Observer Pattern

- Context:
  - The change in one object may trigger operations in one or more other objects
- Problem
  - High coupling
  - Number and type of objects to be notified may not be known in advance

Observer Pattern

- Solution
  - Define a base Subject class that provides methods to
    - Manage observers registrations
    - Send notifications
  - Define a standard Observer interface with a method that receives the notifications

Observer – Consequences

+ Abstract coupling between Subject and Observer
+ Support for broadcast communication
– Unanticipated updates

Flagging interface idiom

- Context:
  - A set of classes is treated similarly but a subset must be treated differently
- Problem:
  - Different objects must be identified at run-time
  - Adding a flag attribute would impact all classes
- Solution:
  - Let different classes implement an empty flagging **`interface`**
  - Check at run-time using **`instanceof`**

Class flagging – Expressions

- Composed expression require parentheses around them
- Implementing Composed flag those classes

```
public interface Composed {/*empty*/}
```

```
public class Operation
implements Expression, Composed {
  // …
}
```

Class flagging – Expression

```
public String formula() {
  String lf = left.formula();
  String rf = right.formula();
  if( left instanceof Composed )
    lf = "(" + lf + ")";
  if( right instanceof Composed)
    rf = "(" + rf + ")";
  return lf +  op + rf;
}
```


Interface **`Cloneable`**

- Implementing **`Cloneable`** flags as safe making a field-for-field copy of instances
- The **`Object.clone()`** method
  - If the class is flagged makes a field-for-field copy of the object
  - Else it throws **`CloneNotSupportedException`**
- By convention, classes that implement this interface should override **`Object.clone()`**
  - Get a copy using **`super.clone()`**
  - Possibly modify fields on the returned object

LAMBDA FUNCTIONS AND METHODS REFERENCES

Ex. anonymous Inner class

```
void process(Object[] v, Consumer p){
  for(Object o : v){
    p.accept(o);
  }
}
```

```
public interface Consumer{
  void accept(Object o);
}
```

Simplified version of `java.util.function.Consumer`

```
Consumer printer = new Consumer(){
  public void accept(Object o){
    System.out.println(o);
  }};
…
```

The only fragment of code really useful. All the rest is just *syntactic sugar*

Lambda function

- Definition of anonymous inner instances for functional interfaces

```
Consumer printer =
              o -> System.out.println(o);
```

```
new Consumer(){
  public void accept(Object o ){
    System.out.println(o);
  }};
```


Lambda expression syntax

parameters **->** *body*

- Parameters
  - None: **()**
  - One: **x**
  - Two or more: **(x, y)**
  - Types can be omitted
    - Inferred from assignee reference type
- Body
  - Expression: **x + y**
  - Code Block: **{ return x + y; }**

Type inference

- Lambda parameter types are usually omitted
  - Compiler can infer the correct type from the context
  - Typically they match the parameter types of the only method in the functional interface

Comparator w/lambda

```
Arrays.sort(sv,
   (a,b) -> ((Student)a).id -((Student)b).id
);
```

Vs.

```
Arrays.sort(sv,new Comparator(){
 public int compare(Object a, Object b){
   return ((Student)a).id -((Student)b).id;
 }});
```

Method reference

- Compact representation of functional interface that invoke single method.

```
Consumer printer = System.out::println;
printer.consume("Hello!");
```

Equivalent to:

```
o -> System.out . println(o);
```

```
new Consumer(){
  public void accept(Object o ){
    System.out . println(o);
}};
```


Method reference syntax

Container :: methodName

Kind	Example
Static method	`Class::staticMethodName`
Instance method of a given object	`object::instanceMethodName`
Instance method of an object of a given type	`Type::methodName`
Constructor	`Class::new`

Static method reference

- Like a C function
  - The parameters are the same as the method parameters

`a,b -> Math.max(a,b)`

```
DoubleBinaryOperator combine = Math::max;
double d=combine.applyAsDouble(1.0, 3.1);
```

```
package java.util.function;
interface DoubleBinaryOperator {
  double applyAsDouble(double a, double b);
}
```


Instance method of object

- Method is invoked on the object
  - Parameters are those of the method

`v -> hexDigits.charAt(v)`

```
String hexDigits = "0123456789ABCDEF";
IntToCharFun hex = hexDigits::charAt;
System.out.println("Hex for 10 : "
                    + hex.apply(10) );
```

```
interface IntToCharFun {
  char apply(int value);
}
```

Instance method reference

- The first argument is the object on which the method is invoked
  - The remaining arguments are mapped to the method arguments

```
s -> s.length()
```

```
StringToIntFunction f = String::length;
for(String s : words){
  System.out.println(f.apply(s));
}
```

```
interface StringToIntFunction {
   int apply(String s);
}
```


Constructor reference

- The return type is a new object
  - Parameters are the constructor's parameters

```
i -> new Integer(i);
```

```
IntegerBuilder builder = Integer::new;
Integer i = builder.build(1);
```

```
interface IntegerBuilder{
     Integer build(int value);
}
```

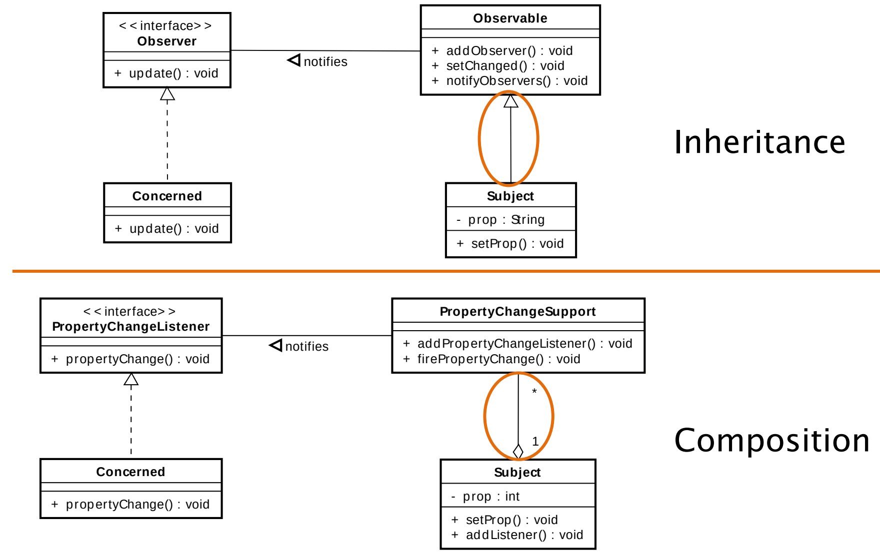
PRACTICAL DESIGN

Inheritance vs. composition

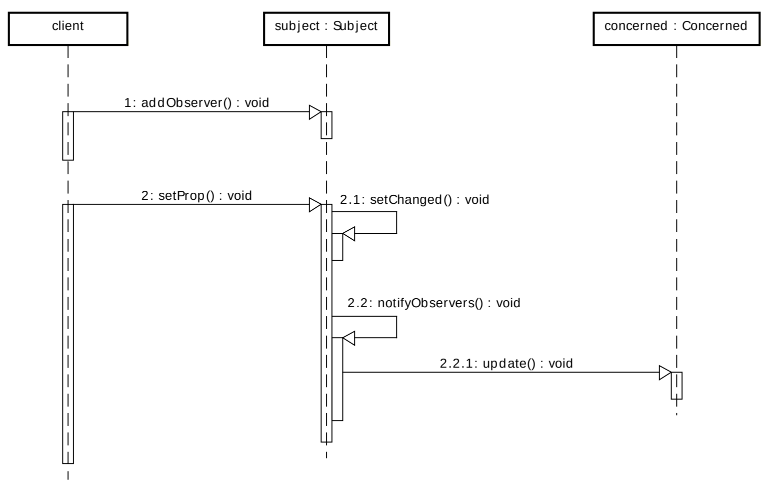
Reuse can be achieved via:

- Inheritance
  - The reusing class inherits reused members that are available as own members
  - Clients can invoke directly inherited methods
- Composition
  - The reusing class has the reused methods available in an included object (attribute)
  - Clients invoke new methods that delegate requests to the included object

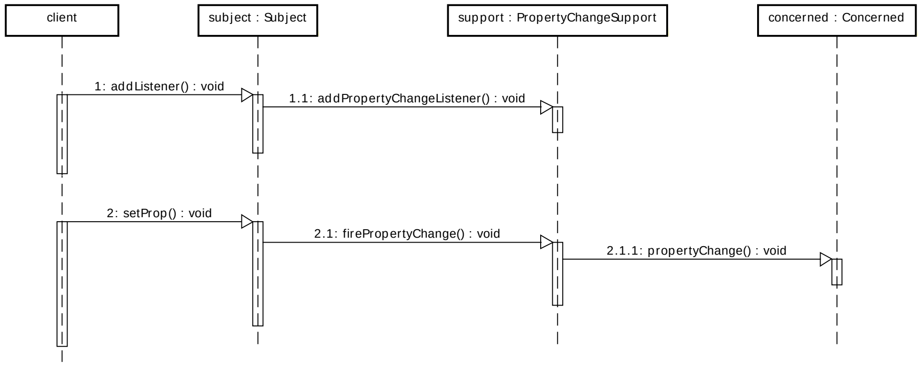
Inheritance vs. Composition



Observer w/Inheritance



Observer w/Composition



Observer subject w/inheritance

```
public class Subject
      extends Observable {

  String  prop="ini";

  public void setProp(String val){
    setChanged();
    property = val;
    notifyObservers("theProp");
  }



}
```

Observer subject w/composition

```
public class Subject {
  PropertyChangeSupport pcs =
        new PropertyChangeSupport(this);
  String  prop=”ini";

  public void setProp(String val) {
    String old = property;
    property = val;
    pcs.firePropertyChange("theProp”,old,val);
  }
  // delegation:
  public void addObs(PropertyChangeListener l){
    pcs.addPropertyChangeListener("theProp”,l);
} }
```

173

Observer with inheritance

```
public class Concerned
     implements Observer {

  @Override
  public void update(Observable src,
                     Object arg) {
    System.out.println("Variation of " +
                       arg);
}

}
```

174

Observer with composition

```
public class Concerned
       implements PropertyChangeListener {

  @Override
  public void propertyChange(
                 PropertyChangeEvent evt) {
    System.out.println("Variation of " +
                      evt.getPropertyName());
}

}
```


Algorithm variability

- Common behavior idiom
  - The variability is bound to the type of objects processed by the algorithm
  - Behavior is implicit in the data classes
  - Less flexibility
- Strategy pattern
  - The variability is implemented through behavioral objects (strategies)
  - Behavior is explicit on algorithm’s invocation
  - More flexibility

Sorting flexibility

```
class Student implements Comparable{ //…
public int compareTo(Object s){
  return id - ((Student)s).id;
}}
```

```
Arrays.sort(students); // <- implicit
```

Natural order

```
// explicit strategy ↓
Arrays.sort(students,(a,b)->{
  return ((Student)a).id - ((Student)b).id;
});
Arrays.sort(students,(a,b)->{
  return ((Student)b).id - ((Student)a).id;
});
```


Wrap-up

- Inheritance
  - Objects defined as sub-types of already existing objects. They share the parent data/methods without having to re-implement
- Specialization
  - Child class augments parent (e.g. adds an attribute/method)
- Overriding
  - Child class redefines parent method
- Implementation/reification
  - Child class provides the actual behaviour of a parent method

Wrap-up

- Polymorphism
  - The same message can produce different behavior depending on the actual type of the receiver objects (late binding of message/method)
- Interfaces provide a mechanism for
  - Constraining alternative implementations
  - Defining a common behavior
  - Behavioral parameterization
- Functional interfaces and lambda simplify the syntax for behavioral parameterization