

Object Oriented Programming

Java

Part 6: Inheritance and polymorphism

Inheritance – revision (1)

- Inheritance is a mechanism where the **subclass** constitute a specialization of a **superclass**. The superclass might be seen as a **generalization** of its subclasses.
- Inheritance is said as a relationship of “**is-a**”.
- Subclasses inherit the attributes and methods of superclasses. The inherit methods might be modified. New attributes and methods might be added to the subclasses.

Inheritance – revision (2)

- **Polymorphism** occurs when there is **redefinition** of methods (methods with the same signature) of the superclass in the subclass.
- In OO, polymorphism is usually implemented through **dynamic binding**, i.e., the method being executed is determined only in runtime (and not in compile time).

Inheritance – revision (3)

- In **simple inheritance** each subclass has only a direct superclass.
- In **multiple inheritance** a subclass might have more than one direct superclass.

Inheritance (1)

- Java adopts the following strategies regarding inheritance:
 - There is only **simple inheritance of classes**.
 - **All classes are subclasses of Object** (explicitly or implicitly).
 - **The subclass inherits all public and protected (but not private) fields and methods of the superclass.**
 - In the case of fields and methods without a visibility modifier (package visibility), the subclass inherits these fields and methods if it is defined in the same package as the superclass, and only in that case.
 - The constructors are not methods, so they are not inherited.

Inheritance (2)

- If the subclass declares a method with the same identifier and parameters (number and type) as one of its superclasses, then the **subclass overrides/redefined that method**.
- If the subclass declares a field with the same identifier as one of its superclasses, then the field of the **subclass hides the field of the superclass** (but it continues to exist!)

Inheritance (3)

Syntax

Modifier* class Ident

```
[ extends IdentC] [ implements IdentI [,IdentI]* ] {  
  [ Fields | Methods ]*  
}
```

- **Modifier**: modifier (visibility, among others)
- **Ident**: class name
- **extends IdentC**: specialization of the superclass
- **implements IdentI**: implementation of interfaces

Inheritance (4)

```
public class SavingAccount extends Account {  
    private static float interestRate=0.05;  
    private long begin;  
    private int interval;  
  
    public void interestRate() {  
        long today=System.currentTimeMillis;  
        if(today==begin+interval) {  
            balance *= (1+interestRate);  
            begin = today;  
        }  
    }  
}
```


Constructors in subclasses (1)

- The subclass constructor should initialize its own fields but only the superclass knows how to correctly initialize its fields.
- The subclass constructor must delegate construction of the inherited fields by either implicitly or explicitly invoking a superclass constructor.
- An **explicit invocation** of a superclass constructor is done with `super()`.
- If the superclass constructor has `N` parameters, these should be passed in the explicit invocation: `super(param1, ..., paramN)`.
- If provided, the explicit invocation must be the first statement in the constructor.

Constructors in subclasses (2)


- The choice of the superclass constructor can be deferred, invoking explicitly one of the class constructors, using `this()` (instead of `super()`).
- If no superclass constructor is called, or if no class constructor is called, as the first statement of a constructor, the no-arg constructor from the superclass is implicitly called before any statement of that constructor.
- If the superclass does not have a no-arg constructor, the superclass constructor invocation is mandatory.

Constructors in subclasses (3)

```
public class A {  
    protected int a;  
    A() {  
        a = 5;  
    }  
    A(int var) {  
        a = var;  
    }  
}
```

It is not necessary to call
explicitly `super()` as
`super()` is implicitly called!

```
public class DuplicateA extends A {  
    DuplicateA() {  
        a *= 2;  
    }  
    DuplicateA(int var) {  
        super(var);  
        a *= 2;  
    }  
}
```



Constructors in subclasses (4)

```
public class A {  
    protected int number;  
    A(int num) {  
        number=num;  
    }  
}
```

It is mandatory to explicitly call `super(-1)` because the superclass does not have a no-arg constructor!

```
public class B extends A {  
    protected String name="not-defined";  
    B() {  
        super(-1);  
    }  
    B(int num) {  
        super(num);  
    }  
    B(int num, String str) {  
        this(num);  
        name= str;  
    }  
}
```

Constructors in subclasses (5)

- When an object is created, memory is allocated for all its fields, including those inherited from superclasses.
- Those fields are set to default initial values for their respective types (zero for all numeric types, false for boolean, '\u0000' for char, and null for object references).
- After this, construction has three phases:
 1. Invoke a superclass constructor (through an implicit or explicit invocation).
 - If the explicit `this()` constructor invocation is used then the chain of such invocations is followed until an implicit or explicit superclass constructor invocation is found.
 - The superclass constructor is executed in the same three phases; this process is applied recursively, terminating when the constructor for `Object` is reached.

Constructors in subclasses (6)

- Any expressions evaluated as part of an explicit constructor invocation are not permitted to refer to any of the members of the current object.
2. Initialize the fields using their initializers and any initialization blocks.
 - In this second stage all the field initializers and initialization blocks are executed in the order in which they are declared.
 - At this stage references to other members of the current object are permitted, provided they have already been declared.
 3. Execute the body of the constructor.

Constructors in subclasses (7)

- When an object of type B is created...

```
public class A {  
    protected int a=1;  
    protected int total;  
    A() {  
        total=a;  
    }  
}
```

```
public class B extends A {  
    protected int b=2;  
    B () {  
        total+=b;  
    }  
}
```

1. Fields with default values
2. Constructor of B is called
3. Constructor of A is called (`super()`)
4. Constructor of `Object` is called
5. Initialization of fields in A
6. Constructor of A is executed
7. Initialization of fields in B
8. Constructor of B is executed

a	b	total
0	0	0
0	0	0
0	0	0
0	0	0
1	0	0
1	0	1
1	2	1
1	2	3

Inheritance and redefinition (1)

- In a subclass it might:
 - be added new fields and methods to the class.
 - be overridden/redefined methods from the superclass.

Inheritance and redefinition (2)

- A subclass method is an **override** of a superclass method if:
 - Both have the same identifier and parameters (number and type).
 - The return type might be **covariant**:
 - If the return type is a reference type then the overriding method can declare a return type that is a subtype of that declared by the superclass method.
 - If the return type is a primitive type, then the return type of the overriding method must be identical to that of the superclass method.

Inheritance and redefinition (3)

- **A method can be overridden only if it is accessible.**
 - If the method is not accessible then it is not inherited, and if it is not inherited it can't be overridden.
 - If a subclass defines a method that coincidentally has the same signature and return type as a superclass private method, they are completely unrelated, the subclass method does not override the superclass private method.
 - Invocations of private methods always invoke the implementation of the method declared in the current class.

Inheritance and redefinition (4)

- The overriding methods have their own access specifies. A subclass can change the access of a superclass method, but only to provide more access.
 - A method declared protected in the superclass can be redeclared **protected** (the usual thing to do) or declared **public**, but it cannot be declared **private** or have **package** access.
 - Making a method less accessible than it was in a superclass violates the contract of the superclass, because an instance of the subclass would not be usable in place of a superclass instance.

Inheritance and redefinition (5)

- **An instance method cannot have the same identifier and parameters (number and type) as an inherited static method, and vice-versa.**
- **The overriding method can, however, be made abstract, even though the superclass method was not.**

Inheritance and redefinition (6)

- **A subclass can change whether a parameter in an overriding method is final** (this is just an implementation detail).
- **The overriding method can be final**, but obviously the method it is overriding cannot.

Inheritance and redefinition (6)

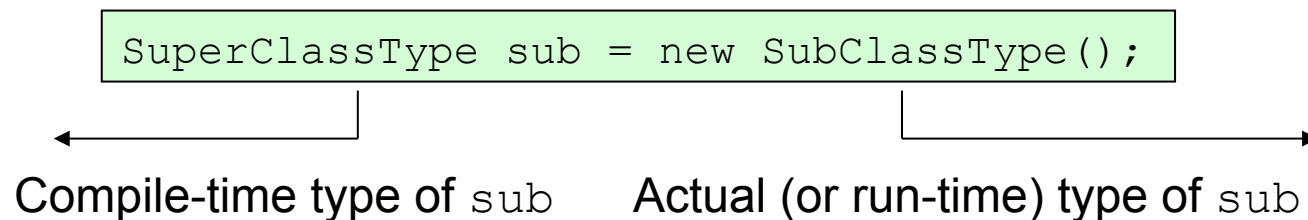
- If a method in the subclass has
 - the same identifier,
 - but different number or parameter types,of a (visible) method in the superclass, then it is an **overloading**.
- If a method in the subclass has
 - the same identifier,
 - the same number or parameter types,
 - but the return is not covariantof a (visible) method in the superclass, then it is a **compile-time error**.

Inheritance and redefinition (7)

- Fields cannot be overridden, they can only be **hidden**.
 - If a field is declared in a subclass with the same name (regardless of type) as one in the superclass, that other field (of the superclass) still exists (in the subclass), but it can no longer be accessed directly by its simple name.
 - In this case, the **super** reference (see slide 32) or another reference of the superclass type must be used to access it.

Polymorphism (1)

- In a class hierarchy, when there is redefinition the implementation of the method is replaced.
 - When an object invokes a method that has been redefined in a subclass, to which method it refers, the superclass method or the subclass method?
 - **When a method is invoked through an object reference, the actual type of the object governs which implementation is used.**



Polymorphism (2)

- **The declared type of a reference is determined in compile time.**
 - The compiler has access to that information directly from the declaration.

```
SuperClass sub = new SubClass();
```

```
SuperClass sub;  
...  
//in any point of the program  
sub = new SubClass();
```

- **Static type:** declared type or explicit cast.

Polymorphism (3)

- The instantiation type (actual or dynamic type) of an object is only determined in run time.
 - The actual type maybe a subclass of the declared type.
 - The instantiation of an object can be done in a different point of the program from the point where the declaration statement was made.
 - Only the program flow will dictate the actual class.

```
SuperClass sub;  
...  
//in other point of the program  
sub = new SubClass();
```

Polymorphism (4)

```
public class SuperClass {  
    protected String str = "SupCStr";  
    public void print() {  
        System.out.println("SupCPrint(): "+str);  
    }  
}
```

```
public class SubClass extends SuperClass {  
    protected String str = "SubCStr";  
    public void print() {  
        System.out.println("SubCPrint(): "+str);  
    }  
    // continues in the next slide
```

Polymorphism (5)

```
// continued from previous slide

public static void main(String[] args) {
    SubClass sub = new SubClass();
    SuperClass sup = sub;
    sup.print();
    sub.print();
    System.out.println("sup.str = "+sup.str);
    System.out.println("sub.str = "+sub.str);
}
}
```

In the terminal is printed

```
SubCImprime(): SubCStr
SubCImprime(): SubCStr
sup.str = SupCStr
sub.str = SubCStr
```

Polymorphism (6)

- Relatively to the previous slide:
 - The declared and instantiation type of `sub` is `SubClass`.
 - The `sub` reference was declared as being of type `SubClass`.
 - Memory was allocated for `sub` as being an object of type `SubClass`.
 - The declared type of `sup` is `SuperClass`.
 - The `sup` reference was declared as being of type `SuperClass`.
 - The instantiation type of `sup` is `SubClass`.
 - Memory was allocated for `sup` as being an object of type `SubClass` (assignment of references, see slide 22 of Java - part 3: methods).
 - The same would happen if

```
sup = new SubClasse();
```

The `super` reference (1)

- The **super** keyword is available in all non-static methods of a class.
- In field access and method invocation, `super` acts as a reference to the current object as an instance of its superclass.
 - Using `super` is the only case in which the type of the reference governs selection of the method implementation to be used.
- An invocation of `super.method` uses always the implementation of the method the superclass defines (or inherits).
 - It does not use any overriding of that method further down the class hierarchy.

The super reference (2)

```
public class SuperClass {  
    protected void name() {  
        System.out.println("SuperClass");  
    }  
}
```

```
public class SubClass extends SuperClass {  
    protected void name() {  
        System.out.println("SubClass");  
    }  
    // continues in the next slide
```

The super reference (3)

```
// continued from previous slide

protected void printName() {
    SuperClass sup = (SuperClass) this;
    System.out.print("this.name(): ");
    this.name();
    System.out.print("sup.name(): ");
    sup.name();
    System.out.print("super.name(): ");
    super.name();
}
}
```

In the terminal is printed

```
this.name(): SubClass
sup.name(): SubClass
super.name(): SuperClass
```


Static members (1)

- **Static members within a class, whether fields or methods, cannot be overridden, they are always hidden.**
- If a reference is used to access a static member then, as with instance fields, static members are always accessed based on the declared type of the reference, not the actual type of the object referred to.

Static members (2)

```
public class SuperClass {  
    protected static String str = "SupCStr";  
    public static void print() {  
        System.out.println("SuperCPrint(): "+str);  
    }  
}
```

```
public class SubClass extends SuperClass {  
    protected static String str = "SubCStr";  
    public static void print() {  
        System.out.println("SubCPrint(): "+str);  
    }  
    // continues in the next slide
```

Static members (3)

```
// continued from previous slide

public static void main(String[] args) {
    SubClass sub = new SubClass ();
    SuperClass sup = sub;
    sup.print();
    sub.print();
    System.out.println("sup.str = "+sup.str);
    System.out.println("sub.str = "+sub.str);
}
}
```

In the terminal is printed

```
SupCPrint(): SupCStr
SubCPrint(): SubCStr
sup.str = SupCStr
sub.str = SubCStr
```

Explicit conversion (1)

- A cast is used to inform the compiler that the casted expression should be interpreted as being declared by the type specified by the cast.
 - **Upcast**: cast of a class to any superclass type (from subclass to superclass, not necessarily a direct superclass).
 - **Downcast**: cast of a class to any subclass type (from superclass to subclass, not necessary a direct subclass).
- The upcast is also known as **safe cast**, because it is always valid.

Explicit conversion (2)

- In Java, a field or local variable declared with a superclass type can refer any actual subclass object. However, it can only be directly:
 - invoked methods of the superclass.
 - accessed fields declared in the superclass.

```
public class A {  
    void foo() {...}  
    ...  
}
```

```
public class B extends A {  
    void b() {...}  
    ...  
}
```

```
public class Warehouse {  
    A var[] = new A[2];  
    void xpto(){  
        var[0] = new A();  
        var[0].foo();  
        var[1] = new B();  
        var[1].foo();  
    }  
}
```

Explicit conversion (3)

- Relatively to the previous slide:
 - References `var[0]` and `var[1]` do not have (direct) access to method `b()` of `B` (even being `var[1]` instantiated as an object of type `B`).
 - However, `var[1]` can invoke method `b()` with a downcast:
`((B)var[1]).b();`
 - On the other hand, the statement `((B)var[0]).b();` is valid in compile time, but it throws an exception in runtime!
 - If the subclass `B` overrides the method `foo()`, which implementation will be invoked with `var[1].foo()`?
 - The actual type of `var[1]` governs the method that will be invoked, in this case the `foo()` from `B` (see slide 24).