Object-oriented programming
Java

Wojciech Complak

Institute of Computing Science
Faculty of Computing
Poznan University of Technology

e-mail: Wojciech.Complak@wsb.poznan.pl

0.9

1

Lecture content

- types
- operators
- statements
- classes
- interfaces

2

The origin and history of *Java*

- 1991: language concept, *Sun Microsystems Inc.*: James Gosling, Patrick Naughton, Chris Warth, Ed Frank, Mike Sheridan; *Oak* programming language, goal: simple, portable programming language for home (electronic) appliances (originally designed for interactive television),
- 1995: *Java* 1.0, first official release, „*Write Once, Run Anywhere*" (*WORA*)
- based on *C++* programming language syntax (indirectly on *C*),
- five principles:
  1. simple, object oriented,
  2. robust and secure,
  3. architecture-neutral and portable
  4. execute with high performance,
  5. interpreted, threaded, and dynamic
- 2010: acquisition of *Sun Microsystems* by *Oracle Corporation*,
- 2019: Java 13

3

Keywords (#1/4)

keyword	meaning
`abstract`	abstract method/class
`assert`	assertion
`boolean`	logical type
`break`	exit from block/loop/*switch* statement (partially replaces *goto*)
`byte`	integer type
`case`	*switch* statement branch
`catch`	exception handling block
`char`	character type
`class`	class declaration
`const`	<reserved, unused>
`continue`	loop continuation
`default`	default *switch* statement branch/implementation in interface
`do`	part of *do-while* loop statement

4

Keywords (#2/4)

keyword	meaning
`double`	floating-point type
`else`	part of *if* conditional statement
`enum`	enumerated type
`extends`	base class extension
`final`	read-only variable, final method/class
`finally`	*catch* statement block
`float`	floating-point type
`for`	*for* loop header
`goto`	<reserved, unused>
`if`	part of *if* conditional statement
`implements`	interface implementation/inherits from
`import`	class/interface import to current namespace
`instanceof`	object type test

5

Keywords (#3/4)

keyword	meaning
`int`	integer type
`interface`	interface declaration
`long`	integer type
`native`	method is implemented in native code (*JNI*)
`new`	new reference type object (array/object)
`package`	package declaration
`private`	access modifier – private member
`protected`	access modifier – protected member
`public`	access modifier – protected member
`return`	return from a method
`short`	integer type
`static`	static member
`strictfp`	restricts floating-point calculations to ensure portability (*Write-Once-Get-Equally-Wrong-Results-Everywhere*)

6

Keywords (#4/4)

keyword	meaning
`super`	access to base class members
`switch`	part of *switch* statement
`synchronized`	method level mutual exclusion
`this`	reference to the current instance of the class
`throw`	throws an exception
`throws`	method may throw declared exception
`transient`	field should not be serialised
`try`	part of exception handling statement
`void`	method does not return value (=procedure)
`volatile`	volatile variable (read from main memory/share by threads)
`while`	part of *while* loop statement
`true, false, null`	reserved

Primitive types (#1/4)

Primitive data types:

- integer (*byte*, *short*, *int*, *long*),
- floating-point (*float*, *double*),
- character (*char*),
- logical (*boolean)*

➢ primitive types (for performance reasons) are not objects (but their object counterparts exist)

✓ in C# primitive types are structures,
✓ Java does not support structures

Primitive types (#2/4)

integer types:

name	description
`byte`	1 byte signed integer, range: -128 (-2^7) to 127 (2^7-1)
`short`	2 byte signed integer, range: -32768 (-2^{15}) to 32767 (2^{15}-1)
`int`	4 byte signed integer, frequently used integer type, range: -2 147 483 648 (-2^{31}) to 2 147 483 647 (2^{31}-1)
`long`	8 byte signed integer, range: –9 223 372 036 854 775 808 (-2^{63}) to 9 223 372 036 854 775 807 (2^{63}-1)

➢ Java does not support unsigned integers

Primitive types (#3/4)

floating-point types (corresponding to IEEE-754 standard):

name	description
`float`	4 byte single precision type, 7 digits precision, approximate range: $\pm 1{,}4 \cdot 10^{-45}$ to $\pm 3{,}4 \cdot 10^{38}$
`double`	8 byte double precision type, 15-16 digits precision, approximate range: $\pm 4.9 \cdot 10^{-324}$ to $\pm 1{,}8 \cdot 10^{308}$

Primitive types (#4/4)

logical type *boolean* has one of two possible values:

- *true* = logical truth
- *false* = logical false

logical values have no arithmetic interpretation (contrary to C/C++, Python)

Naming conventions

- classes – nouns, *PascalCase*, avoid acronyms and abbreviations (unless the abbreviation is more widely used than the long form, e. g URL or HTML),
- interfaces – adjectives, should end with „able/ible”, *PascalCase*, avoid acronyms and abbreviations (unless the abbreviation is more widely used than the long form, e. g URL or HTML),
- methods – should contain a verb, *camelCase*, may contain adjectives and nouns,
- variables – *camelCase*, should not contain ‚_’ and ‚$’, mnemonic, one-character variable names should be avoided except for temporary "throwaway" variables (loop counters, variables in *catch* branches ...),
- constants – all uppercase letters, multiple words separated by ‚_’,
- packages – lowercase letters, syntax corresponding to domain names.

Elementary program structure

hello package

```
package hello;
class Hello
{
  public static void main(String[] args)
  {
     System.out.println("Hello");
  }
}
```

Hello class
(Hello.java file)

main program method
(every class may contain *main()* method)

println method outputs text in console mode to *out* stream of *System* class

Compiling and executing programs

- compilation:

`Hello.java` → javac Hello.java → `Hello.class`

- ✓ source file is a so-called compilation unit,
- ✓ usually contains single class/interface,
- ✓ filename (and path) must (including casing !) comply with its contents – compiling and executing program depends on it

- program execution:

java Hello

Operators (#1/2)

=	assignment
+	addition (unary/binary)
-	subtraction (unary/binary)
*	multiplication
/	division (integer)
%	division (remainder)
++	incrementation
--	decrementation
==	comparison (equal)
!=	comparison (inequal)
<	comparison (less than)
<=	comparison (less or equal)
>	comparison (greater than)
>=	comparison (greater or equal)

!	logical NOT
&&	logical AND (short evaluation)
&	logical AND (full evaluation)
||	logical OR (short evaluation)
|	logical OR (full evaluation)
~	bitwise complement (NOT)
&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	bitwise left shift
>>	bitwise right shift
>>>	bitwise unsigned (zero-fill) right shift
op=	compound assignment (+=, -=, *=, /=, %=, &=, |=, ^=,<<=, >>=)

Operators (#2/2)

- the conditional logical *AND* operator (&&) and the conditional logical *OR* operator (||) require logical operands (boolean) – contrary to C/C++
- operands of *AND* operator (&&) and *OR* operator (||) are evaluated according to short-circuit evaluation principle (minimal/McCarthy/lazy evaluation), arguments are evaluated left-to-right, the second argument is executed/evaluated only if the first argument does not suffice to determine the value of the expression, e.g.:

```
boolean a1, b1;
...
a1 = true;
if (a1 || b1) ... // b1 will never be evaluated
```

- if (for example because of side effect(s)) standard behaviour of the *AND* operator and *OR* operator is required conditional logical operators & and | always evaluate both operands

```
boolean a1, b1;
...
a1 = true;
if (a1 | b1)  ... // b1 will always be evaluated
```


Code block

a code block allows to place several statements in a place where one statement is expected according to the syntax:

`statement;` ⇒

```
{
  statement1;
  statement2;
  …
  statementn;
}
```

example:

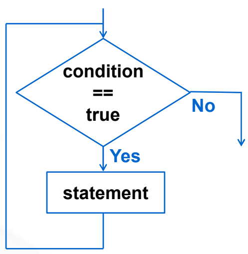
```
{
  System.out.print("Hello ");
  System.out.println("World!");
}
```


while loop (#1/2)

syntax:

```
while(boolean_expression)
  statement;
```

- the condition is tested before executing the loop,
- the body of the loop may never be executed



while loop (#2/2)

example:

```
int n = 5;
while (n>0) {
    System.out.println(n);
    n--;
}
```

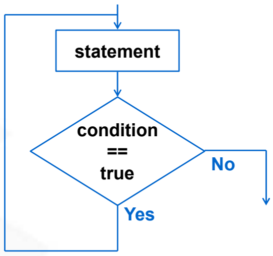
```
5
4
3
2
1
```


do-while loop (#1/2)

syntax:

```
do
  statement;
while(boolean_expression);
```

- the condition is tested **after** execution of the loop body
- the body of the loop must be executed at least once



do-while loop (#2/2)

example:

```
int n = 5;
do
    System.out.println(n);
while (--n > 0);
```

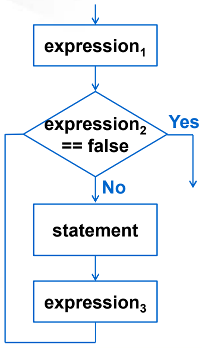
```
5
4
3
2
1
```


for loop (#1/3)

syntax:

$$\texttt{for(expression}_1\texttt{; expression}_2\texttt{; expression}_3\texttt{) instrukcja;}$$

expression$_1$ – **initialization**
expression$_2$ – **continuation condition**
expression$_3$ – **modifier**



for loop (#2/3)

syntax:

$$\texttt{for(expression}_1\texttt{; expression}_2\texttt{; expression}_3\texttt{) statement;}$$

- any (all) expressions may be omitted,
- *expression$_1$* is the initializing part of the loop, it is computed only once; if it is a compound expression, individual component expressions are separated by commas,
- *expression$_2$* is the condition of continuation of the loop, if omitted it is equivalent to constant *true* (the condition is always *true*),
- *expression$_3$* is computed after each loop run and defines the change in the loop state; if it is a compound expression the individual parts are separated by commas.

for loop (#3/3)

example:

```
for (int i = 1, j = 2;
     j < 1000;
     i++, j *= i)
   System.out.println("[" + i + "," + j + "]");
```

```
[1,2]
[2,4]
[3,12]
[4,48]
[5,240]
```

break and *continue* statements

- the *break* statement is used to terminate the loop execution (*for*, *while*, *do-while*)
- the *continue* statement (much rarer) is used to interrupt the current run of the loop and transfer control back to the beginning of the loop (*for*, *while*, *do-while*)

```
while(...)
{
  ...
  break;
  ...
  continue;
  ...
}
...
```

```
do
{
  ...
  break;
  ...
  continue;
  ...
}
while(...);
...
```

```
for(...;
    ...;
    ...)
{
  ...
  break;
  ...
  continue;
  ...
}
...
```

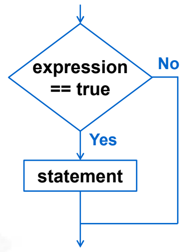

if conditional statement (#1/2)

syntax:

```
if(expression)statement;
```

example:

```
if ((n % 2) == 0)
    System.out.println("Even");
```



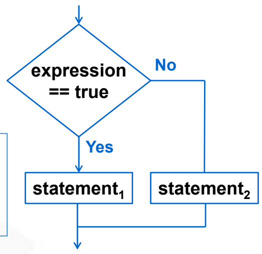
if conditional statement (#2/2)

syntax:

```
if(expression)statement1;
else statement2;
```

example:

```
if ((n % 2) == 0)
    System.out.println("Even");
else System.out.println("Odd");
```



switch statement (#1/2)

- *switch* statement causes the transfer of control to one of the control branches depending on the value of the expression (integer: *byte*, *short*, *int*, not *long;* char, enumerated or *String* type)
- branches of the switch statement are compound instructions (it is not necessary to embed them in a code block {})
- each branch can be labelled with one or more *case* labels
- in a switch statement all *case* constants must have different values
- if none of the constant cases is equal to the value of the control expression, the *default* branch (if present) is executed
- *break* statement terminates branch (and *switch*) execution

switch statement (#2/2)

example:

```
switch (j) {
    case 0:
    case 1:
        System.out.println("less than 2");
    case 2:
    case 3:
        System.out.println("less than 4");
        break;
    default:
        System.out.println("greater than 4");
}
```


goto statement

goto keyword is reserved but not used,

in Java *break* can be used to exit nested loop/block

`break block_label`

(jump forward only)

```
loop_label: for(...; ...; ...)
{
  ...
  block_label: {
     ...
     if(...)break loop_label;
     ...
     if(...)break block_label;
     ...
  }
  ...
}
...
```

Object-oriented programming

Class definition

```
public class My2dPoint {
  int x, y;
  void movePoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
```

class visibility:

- default visibility: (no modifier) – accessible only within the current *package*, inaccessible from outside,

or

- *public* visibility – accessible within and from outside the current *package*

31

Object-oriented programming

Class definition

```
public class My2dPoint {
  int x, y;
  void movePoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
```

fields (attributes)
and
methods
are by default visible only within the current package

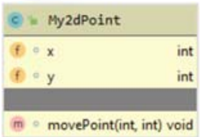
modifier	class	package	subclass	others
public	Y	Y	Y	Y
protected	Y	Y !	Y	N
no modifier	Y	Y !	N	N
private	Y	N	N	N

32

Object-oriented programming

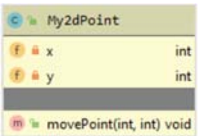
Class definition

```
public class My2dPoint {
  int x, y;
  void movePoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
```



encapsulation

```
public class My2dPoint {
  private int x, y;
  public void movePoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
```



33

Object-oriented programming

Creating objects

```
public class My2dPoint {
  private int x, y;
  public void movePoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
```

```
My2dPoint t;
```

class is a reference type, this statement does not create object, it merely defines a reference to objects of the class (contrary to C++)

```
t = new My2dPoint();
```

object is created now

```
My2dPoint t = new My2dPoint();
```

reference declaration and object creation in one statement

34

Object-oriented programming

this reference (#1/2)

```
public class My2dPoint {
  private int x, y;
  public void movePoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
```

the "this" keyword is used to refer to the current instance (object) of the class,
Smalltalk, Object Pascal → *self*,
C++, Java → *this*,
Visual Basic → *Me*

```
public class My2dPoint {
  private int x, y;
  public void movePoint(int newX, int newY)
  {
    this.x = newX;
    this.y = newY;
  }
}
```

some coding conventions (implemented in static code analysis tools) recommend using *this* reference even if it is not required

35

Object-oriented programming

this reference (#2/2)

```
public class My2dPoint
{
  private int x, y;
  public void movePoint(int newX, int newY)
  {
    this.x = newX;
    this.y = newY;
  }
}
```

this reference is necessary to differentiate between the method parameter and class field/property if they both have the same name

```
class My2dPoint
{
  My2dPoint Operation1(int param)
  {
    /* perform op(s) */
    return this;
  }
}
```

- *this* reference may also be useful to access the class instance (object) from outside of it,
- returning *this* from methods supports a programming technique called *method chaining* (a *fluent interface*) providing better readability of the source code close to that of ordinary written prose (this API design pattern was first coined by Eric Evans and Martin Fowler)

36

Encapsulation

```
class My2dPoint
{
    private int x;
    private int y;
    public void setX(int newX) {      // value verification
        this.x = newX;
    }
    public int getX() {
        return this.x;
    }
    public void setY(int newY) {      // value verification
        this.y = newY;
    }
    public int getY() {
        return this.y;
    }
}
```

encapsulated fields

mutator

accessor

mutator

accessor

37

Creating objects: constructor

- constructor = method with the same name as the class (C++/Java/C#, in Delphi the method marked with the *Constructor* keyword),
- the constructor is automatically started when creating the object (instance of the class) to initialize the object (what the constructor exactly does depends on the language and implementation),
- the task of the constructor is to make the object usable,
- a class can have any number of constructors,
- the constructor can have any set of parameters (parameterless = default),
- the constructor cannot return a result,
- the constructor signals errors by throwing exceptions

38

Default constructor: auto-implemented (#1/2)

a "default constructor" refers to a nullary public constructor automatically generated by the compiler if no constructors have been defined for the class. The default constructor implicitly calls the superclass's nullary constructor and initialises fields to default values.

```
class My2dPoint
{
  private int x, y;
}
// ...
My2dPoint t = new My2dPoint();
```

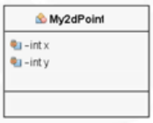
calling the default constructor

in the absence of a public default constructor, it wouldn't be possible to directly create class objects

39

Default constructor: auto-implemented (#2/2)

```
class My2dPoint
{
  private int x, y;
}
```



the class diagram does NOT show the automatically generated default constructor

40

Parameterless constructor: explicit (#1/2)

user-defined default constructor

```
class My2dPoint {
    private int x;
    private int y;
public My2dPoint() {
    this.x = 0;
    this.y = 0;
    }
}
// ...
My2dPoint t = new My2dPoint();
```

default constructor implementation

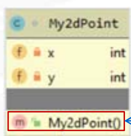
invocation of the default constructor

☞ if the default constructor is the only implemented, it must be (package) *public*, otherwise (*private*) you will not be able to create objects of this class,

41

Parameterless constructor: explicit (#2/2)

```
class My2dPoint {
    private int x;
    private int y;
public My2dPoint() {
    this.x = 0;
    this.y = 0;
    }
}
```



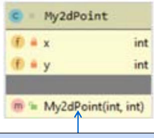
the class diagram DOES show the user-defined parameterless constructor

42

Object-oriented programming

Constructors (#1/5)

user-defined constructor




```
class My2dPoint {
  private int x;
  private int y;
  public My2dPoint(int newX, int newY) {
      this.x = newX;
      this.y = newY;
  }
}
// ...
My2dPoint t1 = new My2dPoint();
My2dPoint t2 = new My2dPoint(1,2);
```

constructor implementation

it's not possible any more – if the programmer defines any custom constructor then Java does not automatically generate a parameterless constructor

calling user-defined constructor

Object-oriented programming (43/87)

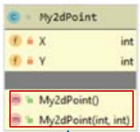
43

Object-oriented programming

Constructors (#2/5)

you can define as many constructors as you like - so that you can conveniently create class objects

```
class My2dPoint {
    private int X;
    private int Y;
public My2dPoint() {
    this.X = 0;
    this.Y = 0;
  }
public My2dPoint(int newX, int newY) {
    this.X = newX;
    this.Y = newY;
  }
}
// ...
My2dPoint t1 = new My2dPoint();
My2dPoint t2 = new My2dPoint(1,2);
```



2 constructors

so now you can...

and also

Object-oriented programming (44/87)

44

Object-oriented programming

Constructors (#3/5)

Java does not support default parameters – duplicating constructors is necessary

```
class My2dPoint
{
   private int x, y;
   public My2dPoint() { x = y = 0; }
   public My2dPoint(int newX) { x = newX; y = 0; }
   public My2dPoint(int newX, int newY) { x = newX; y = newY; }
}
// ..
My2dPoint a = new My2dPoint();        x:0,y:0
          a = new My2dPoint(1);       x:1,y:0
          a = new My2dPoint(1,2);     x:1,y:2
```

Object-oriented programming (45/87)

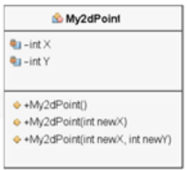
45

Object-oriented programming

Constructors (#4/5)

Java does not support default parameters – duplicating constructors is necessary

```
public class My2dPoint
{
  private int X, Y;
  public My2dPoint()
  {
     this(0,0);
  }
  public My2dPoint(int newX)
  {
     this(newX,0);
  }
  public My2dPoint(int newX, int newY)
  {
     this.X = newX;
     this.Y = newY;
  }
}
```



Object-oriented programming (46/87)

46

Object-oriented programming

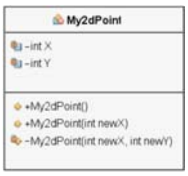
Constructors (#5/5)

a ctor can be *private* - it can still be invoked by other constructors but it cannot be used directly to create an instance of the class

```
public class My2dPoint
{
  private int X, Y;
  public My2dPoint() { this(0,0); }
  public My2dPoint(int newX) { this(newX,0); }
  private My2dPoint(int newX, int newY)
  {
      this.X = newX;
      this.Y = newY;
  }
}
// ...
My2dPoint t1 = new My2dPoint();
My2dPoint t2 = new My2dPoint(1,2);
```

so this is valid ...

and this is not ...



Object-oriented programming (47/87)

47

Object-oriented programming

UML

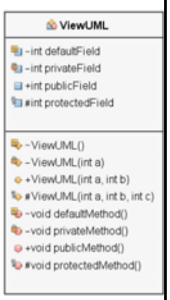
class members

```
public class ViewUML
{
  int defaultField;
  private int privateField;
  public int publicField;
  protected int protectedField;

  ViewUML() {}
  private ViewUML(int a) {}
  public ViewUML(int a, int b) {}
  protected ViewUML(int a, int b, int c) {}

  void defaultMethod() {}
  private void privateMethod() {}
  public void publicMethod() {}
  protected void protectedMethod() {}
}
```

class fields

constructors

methods



Object-oriented programming (48/87)

48

Static members (#1/3)

static modifier associates a member with the class, rather than with an object/instance

```
class Test
{
   static private int field1 = 5;
   static public int field2 = 6;
   private int field3 = 4;
   static void classMethod()
   {
      field1 += field2;
      // field3++;
      // instanceMethod();
   }
   void instanceMethod ()
   {
      field1 += field2;
      this.field3++;
      classMethod();
   }
}
```

static fields private and public

object/instance field

static method

instance (non-static) method

- static members can be used even when no instance of the class exists,
- static field declaration creates one instance of a variable shared by all class objects,
- static fields are created when the program is started

Static members (#2/3)

static modifier associates a member with the class, rather than with an object/instance

```
class Test
{
   static private int field1 = 5;
   static public int field2 = 6;
   private int field3 = 4;
   static void classMethod()
   {
      field1 += field2;
      // field3++;
      // instanceMethod();
   }
   void instanceMethod ()
   {
      field1 += field2;
      this.field3++;
      classMethod();
   }
}
```

static fields private and public

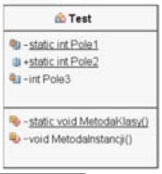
object/instance field

static method has access only to static members, it cannot access instance members (because no class object may exist), therefore *this* cannot be used

instance (non-static) method has access to static (they always exist) and non-static members

Static members (#3/3)

access to members




```
class Test
{
   static private int field1 = 5;
   static public int field2 = 6;
   private int field3 = 4;
   static void classMethod() { }
   void instanceMethod () { }
}
// ...
Test.classMethod();

Test t = new Test();
t.instanceMethod();
```

use the class name to access static members - regardless of whether any class object exists

use the reference to an object of the class to access non-static members

static class

static class contains only static members (☞ there is no need to keep the object state) – Java supports only nested static classes – simulating a global static class requires:
- prohibit inheritance (*final*)
- hide default constructor (private)
- flag all members (*static*)

```
final class Test
{
   private Test() {}
   static private int field1 = 5;
   static public int field2 = 6;
   static void instanceMethod()
   {
      /* ... */
   }
}
// ...
Test.Method1();
Test t = new Test();
```

☞ an object of a static class cannot be created,
☞ all members of static class must be explicitly declared *static* (by default members are non-static

use the class name to access static members

an object of a static class cannot be created

Static block

static block is necessary to initialise static fields

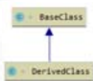
```
class Test {
   static private int field1;
   static public int field2;
   private int field3 = 4;
   static {
      field1 = 0;
      field2 = 0;
      // field3 = 0;
   }
   public Test()  {
      field1 = 0;
      field2 = 0;
      this.field3 = 0;
   }
}
```

static block:
- run once at class start-up,
- has access only to static components,
- cannot have any access modifiers
- it cannot be called directly (by the programmer)

instance constructor:
- run when creating/initialising objects,
- has access to both static and non-static components,
- can have access modifiers
- unlimited number of instance constructors, each has different set of arguments,

Inheritance "is-a" (#1/6)

creating class hierarchy



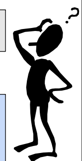

```
class BaseClass {
  }

class DerivedClass extends BaseClass {
  }
```

base class

derived class

based upon the definition of the base class, we create a derived (child) class that:
☞ has (inherits) all members of the base class (but does not necessarily has access to them)
☞ adds new members
☞ redefines (overrides) the inherited members
☞ the derived class has (usually) more functionality than the base class

this example is correct but ... hardly useful



terminology (depending on the author, language, translation ...):
- *base class = superclass = parent class*
- *derived class = subclass = child class*

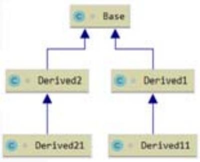
the superclass and subclass are an analogy to the concepts of set theory - the subclass is a subset of the superset

Object-oriented programming

Inheritance "is-a" (#2/6)

existing derived class can become the base class for the next one...

```
class Base {
    }
class Derived1 extends Base {
    }
class Derived11 extends Derived1 {
    }
class Derived2 extends Base {
    }
class Derived21 extends Derived2 {
    }
```



- class *Base* is the root of the tree (it's only the base class),
- classes *Derived1* and *Derived2* are intermediate classes because they are both base classes (for *Derived11* i *Derived21*) and derived from class *Base*,
- classes *Derived11* and *Derived21* are leaves in the tree – only derived classes

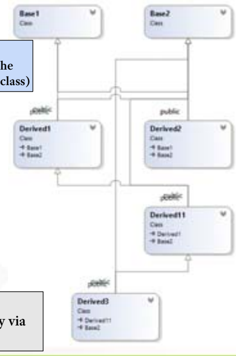
Object-oriented programming

Inheritance "is-a" (#3/6)

C++

```
class Base1
{
};
class Base2
{
};
class Derived1 : public Base1, public Base2
{
};
class Derived2 : public Base1, public Base2
{
};
class Derived11 : public Derived1, public Base2
{
};
class Derived3 : public Derived11, public Base2
{
};
```

important feature of inheritance model of C++ (not supported by C# and Java): the mode of inheriting from the base class (one can voluntarily restrict access to the base class)



- *Derived1* class – multiple inheritance from *Base1 and Base2*
- *Derived3* inherits (among others) twice from *Base2* : indirectly via *Derived11* and directly

Object-oriented programming

Inheritance "is-a" (#4/6)

inheritance: adding new fields

```
class Point {
    private double x;
    private double y;
    }
class Circle extends Point {
    private double r;
    }
```

base class contains two fields: *x, y*

derived class:
- inherits two fields (*x, y*) from its base class
- adds new field (*r*)

„is-a" = Liskov *substitution principle*: objects of a superclass shall be replaceable with objects of its subclasses without breaking the application

Object-oriented programming

Inheritance "is-a" (#5/6)

inheritance: adding new fields and methods

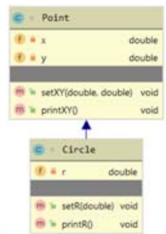
```
class Point {
    private double x;
    private double y;
    public void setXY(double newX, double newY) {
        this.x = newX;
        this.y = newY;
    }
    public void printXY() {
        System.out.println("X:" + this.x + ",Y:" + this.y);
    }
}
class Circle extends Point {
    private double r;
    public void setR(double newR) {
        this.r = newR;
    }
    public void printR() {
        System.out.println("R:" + this.r);
    }
}
```

private instance fields

public instance methods

class inherits fields x,y
adds field r

public instance methods



Object-oriented programming

Inheritance "is-a" (#6/6)

fields and methods: usage

```
class Point {
    private double x; private double y;
    public void setXY(double newX, double newY) { this.x = newX; this.y = newY; }
    public void printXY() { System.out.println("X:" + this.x + ",Y:" + this.y); }
}
class Circle extends Point {
    private double r;
    public void setR(double newR) { this.r = newR; }
    public void printR() { System.out.println("R:" + this.r); }
}
```

```
Point p = new Point(); // create new class Point object
p.setXY(1, 2);  // set private fields x,y in class Point object
p.printXY(); // display Point object fields

Circle C = new Circle(); // create new class Circle object
C.setXY(3, 4); // set class Circle instance fields
               // inherited from Point class, using inherited setXY() method
C.setR(5);     // set private field r in class Circle object
C.printXY();  // use inherited from Point method
C.printR();   // display Circle object field
```

X:1,Y:2

X:3,Y:4
R:5

Object-oriented programming

Inheritance: access to members (#1/9)

access to members

```
class Base {
    int fieldB1;           // package field
    private int fieldB2;   // private field
    public int fieldB3;    // public field
    protected int fieldB4; // protected field
    void methodB1() { }     // package method
    private void methodB2() { } // private method
    public void methodB3() { }  // public method
    protected void methodB4() { } // protected method
}
class Derived extends Base {
    int fieldD1;           // package field
    private int fieldD2;   // private field
    public int fieldD3;    // public field
    protected int fieldD4; // protected field
    void methodD1() { }     // package method
    private void methodD2() { } // private method
    public void methodD3() { }  // public method
    protected void methodD4() { } // protected method
}
```

Object-oriented programming

Inheritance: access to members (#2/9)

access to members

```
class Base {
    int fieldB1;          // package field
```

package (by default) instance fields are:
- accessible for class methods
- inherited and accessible for derived class methods
- accessible within current package

```
    public void methodB3() { }  // public method
    protected void methodB4() { } // protected method
}
class Derived extends Base {
    int fieldD1;          // package field
```

package (by default) instance fields are:
- accessible for class methods
- inherited and accessible for derived class methods
- accessible within current package

```
    public void methodD3() { }  // public method
    protected void methodD4() { } // protected method
}
```

Object-oriented programming (61/87)

61

Object-oriented programming

Inheritance: access to members (#3/9)

access to members

```
class Base {
    int fieldB1;          // package field
    private int fieldB2;  // private field
```

private instance fields are:
- accessible for class methods
- inherited but inaccessible for derived class methods
- inaccessible from outside of the class

```
    protected void methodB4() { } // protected method
}
class Derived extends Base {
    int fieldD1;          // package field
    private int fieldD2;  // private field
```

private instance fields are:
- accessible for class methods
- inherited but inaccessible for derived class methods
- inaccessible from outside of the class

```
    protected void methodD4() { } // protected method
}
```

Object-oriented programming (62/87)

62

Object-oriented programming

Inheritance: access to members (#4/9)

access to members

```
class Base {
    int fieldB1;          // package field
    private int fieldB2;  // private field
    public int fieldB3;   // public field
```

public instance fields are:
- accessible for class methods
- inherited and accessible for derived class methods
- accessible from outside of the class

```
}
class Derived extends Base {
    int fieldD1;          // package field
    private int fieldD2;  // private field
    public int fieldD3;   // public field
```

public instance fields are:
- accessible for class methods
- inherited and accessible for derived class methods
- accessible from outside of the class

```
}
```

Object-oriented programming (63/87)

63

Object-oriented programming

Inheritance: access to members (#5/9)

access to members

```
class Base {
    int fieldB1;          // package field
    private int fieldB2;  // private field
    public int fieldB3;   // public field
    protected int fieldB4; // protected field
```

protected instance fields are:
- accessible for class methods
- inherited and accessible for derived class methods
- inaccessible from outside of the class

```
}
class Derived extends Base {
    int fieldD1;          // package field
    private int fieldD2;  // private field
    public int fieldD3;   // public field
    protected int fieldD4; // protected field
```

protected instance fields are:
- accessible for class methods
- inherited and accessible for derived class methods
- inaccessible from outside of the class

```
}
```

Object-oriented programming (64/87)

64

Object-oriented programming

Inheritance: access to members (#6/9)

access to members

package (by default) methods:
- can access all class members
- are accessible in derived classes
- are accessible within the current package

```
class E
    int
    pri
    pub
    pro
    void methodB1() { }      // package method
    private void methodB2() { } // private method
    public void methodB3() { }  // public method
    protected void methodB4() { } // protected method
}
cl
```

package (by default) methods :
- can access all class members
- can access public and protected members of (directly) base class
- are accessible in derived classes
- are accessible within the current package

```
    void methodD1() { }      // package method
    private void methodD2() { } // private method
    public void methodD3() { }  // public method
    protected void methodD4() { } // protected method
}
```

Object-oriented programming (65/87)

65

Object-oriented programming

Inheritance: access to members (#7/9)

access to members

```
class Base {
    int fiel
    private
    public i
    protecte
    void met
```

private methods:
- can access all class members
- are inaccessible in derived classes
- are inaccessible from outside the class

```
    private void methodB2() { } // private method
    public void methodB3() { }  // public method
    protected void methodB4() { } // protected method
}
class D
    int
    pri
    pub
    pro
    voi
```

private methods :
- can access all class members
- can access public and protected members of (directly) base class
- are inaccessible in derived classes
- are inaccessible from outside the class

```
    private void methodD2() { } // private method
    public void methodD3() { }  // public method
    protected void methodD4() { } // protected method
}
```

Object-oriented programming (66/87)

66

Object-oriented programming

Inheritance: access to members (#8/9)

access to members

```
class Base {
    int fieldB1;            // package field
    private
    public i
    protecte
    void met
    private
    public void methodB3() { }  // public method
    protected void methodB4() { } // protected method
}
class Derived extends Base {
    int
    priv
    publ
    prot
    voic
    priv
    public void methodD3() { }  // public method
    protected void methodD4() { } // protected method
}
```

public methods:
- can access all class members
- are accessible in derived classes
- are accessible from outside the class

public methods :
- can access all class members
- can access public and protected members of (directly) base class
- are accessible in derived classes
- are accessible from outside the class

Object-oriented programming (67/87)

67

Object-oriented programming

Inheritance: access to members (#9/9)

access to members

```
class Base {
    int fieldB1;            // package field
    private int fieldB2;  // private field
    public i
    protecte
    void met
    private
    public v
    protected void methodB4() { } // protected method
}
class Derived extends Base {
    int fieldD1;            // package field
    priv
    publ
    prot
    void
    priv
    publ
    protected void methodD4() { } // protected method
}
```

protected methods:
- can access all class members
- are accessible in derived classes
- are inaccessible from outside the class

protected methods:
- can access all class members
- can access public and protected members of (directly) base class
- are accessible in derived classes
- are inaccessible from outside the class

Object-oriented programming (68/87)

68

Object-oriented programming

Inheritance: constructors

the order of constructors invocation

```
class BaseClass {
    protected BaseClass() {
        System.out.println("Base class constructor");
    }
}
class DerivedClass extends BaseClass {
     public DerivedClass() {
        System.out.println("Derived class constructor");
    }
}
// ...
DerivedClass p1 = new DerivedClass();
```

Base class constructor
Derived class constructor

Object-oriented programming (69/87)

69

Object-oriented programming

Inheritance: constructors

visibility of base class constructors

```
class Base
{
  ????? Base() { }
}
```

- *<none> - public* within package
- *public* – the object can be created both directly and by a child object,
- *protected* – the object can be created by child object(s),
- *private* – you cannot create an object of this class (and hence a child) (☞Singleton)

Object-oriented programming (70/87)

70

Object-oriented programming

Inheritance: field overriding (#1/2)

overriding members

```
class Test {
    protected int a;
}
class Test1 extends Test {
    private int a;
    public void modifyA(int a) {
        this.a = 5;
        super.a = 4;
        System.out.println(this.a + " " + super.a + " " + a);
        }
}
```

method parameter overrides field,
instance field overrides inherited field

if there is a larger difference in levels, you must use intermediate methods (as opposed to C++)

```
Test1 t1 = new Test1();
t1.modifyA(1);
```

5 4 1

Object-oriented programming (71/87)

71

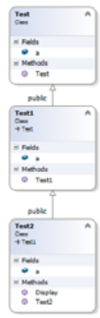
Object-oriented programming

Inheritance: field overriding (#2/2)

access to overridden members

C++

```
class Test
{
 public:
  int a;
  Test() { this->a = 1; }
};
class Test1 : public Test
{
 public:
  int a;
  Test1() { this->a = 2; }
};
class Test2 : public Test1
{
 public:
  int a;
  Test2() { this->a = 3; }
  void Display() {
    cout << this->a << Test1::a << Test::a << endl;
  }
};
```



Object-oriented programming (72/87)

72

Inheritance: method overriding

member overriding

```
class Test {
  public void speak() {
    System.out.println("Class Test method invoked on behalf of object:" + this);
    }
}
class Test1 extends Test {
  public void speak() {
    super.speak();
    System.out.println("Class Test1 method invoked on behalf of object:" + this);
    }
}
```

- method overrides (no keyword!) inherited method
- if there is a larger difference in levels, you must use intermediate methods (as opposed to C++)

```
Test t = new Test();
t.speak();
Test1 t1 = new Test1();
t1.speak();
```

```
Class Test method invoked on behalf of object:Test@15db9742
```

```
Class Test method invoked on behalf of object:Test1@6d06d69c
Class Test1 method invoked on behalf of object:Test1@6d06d69c
```


73

Polymorphism

overriding methods (#1/6)

```
class A {
    public void speak() {
        System.out.println("class A method invoked on behalf of object " + this);
        }
}
class B extends A {
    public void speak() {
        super.speak();
        System.out.println("class B method invoked on behalf of object " + this);
        }
}
class C extends B {
    public void speak() {
        super.speak();
        System.out.println("class C method invoked on behalf of object " + this);
        }
}
// ...
A a = new A(); a.speak();

B b = new B(); b.speak();
a = b;         a.speak();

C c = new C(); c.speak();
a = c;         a.speak();
b = c;         b.speak();
```

class A method invoked on behalf of object Test.A@1b6d3586

74

Polymorphism

overriding methods (#2/6)

```
class A {
    public void speak() {
        System.out.println("class A method invoked on behalf of object " + this);
        }
}
class B extends A {
    public void speak() {
        super.speak();
        System.out.println("class B method invoked on behalf of object " + this);
        }
}
class C extends B {
    public void speak() {
        super.speak();
        System.out.println("class C method invoked on behalf of object " + this);
        }
}
// ...
A a = new A(); a.speak();

B b = new B(); b.speak();
a = b;         a.speak();

C c = new C(); c.speak();
a = c;         a.speak();
b = c;         b.speak();
```

class A method invoked on behalf of object Test.B@4554617c
class B method invoked on behalf of object Test.B@4554617c

75

Polymorphism

overriding methods (#3/6)

```
class A {
    public void speak() {
        System.out.println("class A method invoked on behalf of object " + this);
        }
}
class B extends A {
    public void speak() {
        super.speak();
        System.out.println("class B method invoked on behalf of object " + this);
        }
}
class C extends B {
    public void speak() {
        super.speak();
        System.out.println("class C method invoked on behalf of object " + this);
        }
}
// ...
A a = new A(); a.speak();

B b = new B(); b.speak();
a = b;         a.speak();

C c = new C(); c.speak();
a = c;         a.speak();
b = c;         b.speak();
```

class A method invoked on behalf of object Test.B@4554617c
class B method invoked on behalf of object Test.B@4554617c

76

Polymorphism

overriding methods (#4/6)

```
class A {
    public void speak() {
        System.out.println("class A method invoked on behalf of object " + this);
        }
}
class B extends A {
    public void speak() {
        super.speak();
        System.out.println("class B method invoked on behalf of object " + this);
        }
}
class C extends B {
    public void speak() {
        super.speak();
        System.out.println("class C method invoked on behalf of object " + this);
        }
}
// ...
A a = new A(); a.speak();

B b = new B(); b.speak();
a = b;         a.speak();

C c = new C(); c.speak();
a = c;         a.speak();
b = c;         b.speak();
```

class A method invoked on behalf of object Test.C@74a14482
class B method invoked on behalf of object Test.C@74a14482
class C method invoked on behalf of object Test.C@74a14482

77

Polymorphism

overriding methods (#5/6)

```
class A {
    public void speak() {
        System.out.println("class A method invoked on behalf of object " + this);
        }
}
class B extends A {
    public void speak() {
        super.speak();
        System.out.println("class B method invoked on behalf of object " + this);
        }
}
class C extends B {
    public void speak() {
        super.speak();
        System.out.println("class C method invoked on behalf of object " + this);
        }
}
// ...
A a = new A(); a.speak();

B b = new B(); b.speak();
a = b;         a.speak();

C c = new C(); c.speak();
a = c;         a.speak();
b = c;         b.speak();
```

class A method invoked on behalf of object Test.C@74a14482
class B method invoked on behalf of object Test.C@74a14482
class C method invoked on behalf of object Test.C@74a14482

78

Object-oriented programming

Polymorphism

overriding methods (#6/6)

```java
class A {
    public void speak() {
        System.out.println("class A method invoked on behalf of object " + this);
        }
}
class B extends A {
    public void speak() {
        super.speak();
        System.out.println("class B method invoked on behalf of object " + this);
        }
}
class C extends B {
    public void speak() {
        super.speak();
        System.out.println("class C method invoked on behalf of object " + this);
        }
}
// ...
A a = new A(); a.speak();

B b = new B(); b.speak();
a = b;         a.speak();

C c = new C(); c.speak();
a = c;         a.speak();
b = c;         b.speak();
```

class A method invoked on behalf of object Test.C@74a14482
class B method invoked on behalf of object Test.C@74a14482
class C method invoked on behalf of object Test.C@74a14482

79

Object-oriented programming

Overriding *Object* class members

```java
public /*final*/ class Point /*extends Object*/ {
    private /*final*/ int x;
    private /*final*/int y;
    public Point(final int x, final int y)  {
        this.x = x;
        this.y = y;
    }
    public String toString() {
        return "[" + this.x + ", " + this.y + ']';
    }
    public boolean equals(final Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (this.getClass() != other.getClass()) return false; // child classes ?
        return this.x == ((Point) other).x && this.y == ((Point) other).y;
    }
    static final int INT_BITS_NUM = 32;
    public int hashCode() {
        return x ^ (y << (INT_BITS_NUM / 2));
    }
}
```

80

Object-oriented programming

Generics

```java
class GenericStack<StackItem> {
   StackItem[] vector;
   int Top;
   public GenericStack() {
      vector = (StackItem[])new Object[100];
      }
   public GenericStack(int Capacity) {
      vector = (StackItem[])new Object[Capacity];
      }
   public void Push(StackItem s) {
      vector[Top++] = s;
      }
   public StackItem Pop() {
      return vector[--Top];
      }
}
```

restrictions/requirements:
`<StackItem extends Class0 & Interface1>`

81

Object-oriented programming

Interfaces (#1/6)

interface describes the set of services provided by a class

```java
interface Services {
  void f1();
  int f2(int a, int b);
}
```

- interfaces resemble pure abstract classes - they cannot contain fields or implementations, only static constants and abstract (except default, Java 8) methods are allowed
- the class that implements the interface (inherits from the interface) must provide the implementation of all interface components,
- all interface elements are public (no modifiers are allowed),
- class can implement any number of interfaces

- implemented interface members must be explicitly *public*
- the class must implement all interface members,

```java
class Test implements Services {
    public void f1() {
        }
    public int f2(int a, int b) {
        return a + b;
        }
    void f3() {
        }
}
```

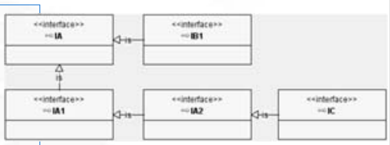
82

Object-oriented programming

Interfaces (#2/6)

interface hierarchies

```java
interface IA {
     }
interface IA1 extends IA {
     }
interface IA2 extends IA1 {
     }
interface IB1 extends IA {
     }
interface IC extends IA2, IB1 {
     }
```



- interfaces support multiple inheritance,
- the same rules for overriding elements as in classes apply

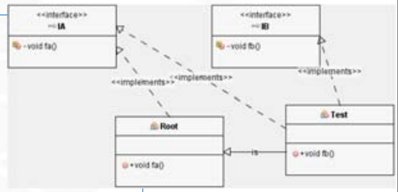
83

Object-oriented programming

Interfaces (#3/6)

inheritance and interfaces implementation

```java
interface IA {
  void fa();
}
interface IB {
  void fb();
}
class Root implements IA {
  public void fa() {
     }
}
class Test extends Root implements IA, IB {
  public void fb() {
     }
}
```



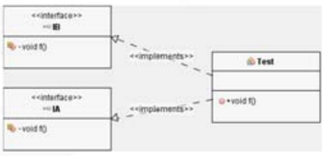
84

Object-oriented programming

Interfaces (#4/6)

interface members name collisions

```
interface IA {
  void f();
}
interface IB {
  void f();
}
class Test implements IA, IB {
  public void f() {
     }
}
```



- one implementation is enough

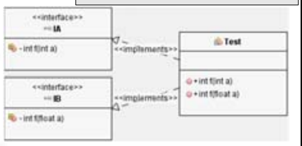
85

Object-oriented programming

Interfaces (#5/6)

interface members name collisions

```
interface IA {
    int f(int a);
}
interface IB {
    int f(float a);
}
class Test implements IA, IB {
    public int f(int a) {
        return 0; }
    public int f(float a) {
        return 0; }
}
```



- polymorphism: overloaded method *f()*,
- difference only in the returned type is not enough (it's not Ada)

86

Object-oriented programming

Interfaces (#6/6)

access to interface members

```
interface ITest {
    void service();
}
class Test implements ITest {
    public void service() {
        System.out.println("Service is run:" + this); }
}
```

```
Test t = new Test();
t.service();

ITest iF = new Test();
iF.service();
```

object level access

any object of a class implementing given interface

87