

INHERITANCE

By Pakita Shamoi

Concept

2

- **Inheritance:** you can create new classes that are built on existing classes. Through the way of inheritance, you can **reuse** the existing class's methods and fields, and you can also add new methods and fields to adapt the new classes to new situations
- Subclass and superclass
- Subclass and superclass have a IsA relationship: an object of a subclass IsA(n) object of its superclass.
- **Example:** *Student (subclass) is a Person (superclass)*

Example

3

Superclass

```
public class Person{
    private String name;

    public Person ( ) {
        name = "no_name_yet";
    }

    public Person ( String initialName ) {
        this.name = initialName;
    }

    public String getName ( ) {
        return name;
    }

    public void setName ( String newName ) {
        name = newName;
    }
}
```

Subclass

```
public class Student extends Person {
    private int studentNumber;

    public Student ( ) {
        super();
        studentNumber = 0;
    }

    public Student (String initialName, int
initialStudentNumber) {
        super(initialName);
        studentNumber = initialStudentNumber;
    }

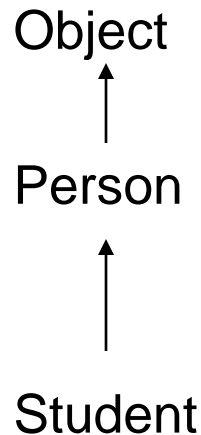
    public int getStudentNumber ( ) {
        return studentNumber;
    }

    public void setStudentNumber (int
newStudentNumber ) {
        studentNumber = newStudentNumber;
    }
}
```

Classes hierarchy

4

- Every class is an extended (inherited) class, whether or not it's declared to be. If a class does not declared to explicitly extend any other class, then it implicitly extends the **Object** class
- Class hierarchy of previous example



Fields/Methods in Extended Classes

5

- An object of an extended class contains two sets of variables and methods
 - ▣ fields/methods which are **defined locally** in the extended class
 - ▣ fields/methods which are **inherited** from the superclass



What are the fields for a **Student** object in the previous example?

Constructors in extended classes

6

- A constructor of the extended class can invoke one of the superclass's constructors by using the *super* method.
- If no superclass constructor is invoked explicitly, then the superclass's no-arg constructor

super ()

is invoked automatically as the first statement of the extended class's constructor.

- *Constructors are not methods and are NOT inherited.*

Using the Keyword `super`

7

The keyword **`super`** refers to the superclass of the class in which **`super`** appears. This keyword can be used in two ways:

- ❑ To call a superclass constructor
- ❑ To call a superclass method

Three phases of an object's construction

8

- When an object is created, memory is allocated for all its fields, which are initially set to be their default values. It is then followed by a three-phase construction:
 - invoke a superclass's constructor
 - initialize the fields by using their initializers and initialization blocks
 - execute the body of the constructor
- The invoked superclass's constructor is executed using the same three-phase constructor. This process is executed recursively until the `Object` class is reached

To Illustrate the Construction Order. . .

9

```
class X {  
    protected int xOri = 1;  
    protected int whichOri;  
    public X() {  
        whichOri = xOri;  
    }  
}
```

```
class Y extends X {  
    protected int yOri = 2;  
    public Y() {  
        whichOri = yOri;  
    }  
}
```

```
}  
Y objectY = new Y();
```

Step	what happens	xOri	yOri	whichOri
0	fields set to default values	0	0	0
1	Y constructor invoked	0	0	0
2	X constructor invoked	0	0	0
3	Object constructor invoked	0	0	0
4	X field initialization	1	0	0
5	X constructor executed	1	0	1
6	Y field initialization	1	2	1
7	Y constructor executed	1	2	2

Superclasses and Subclasses

10

Superclass

Circle

Circle Methods

Circle Data

Inheritance

Subclass

Cylinder

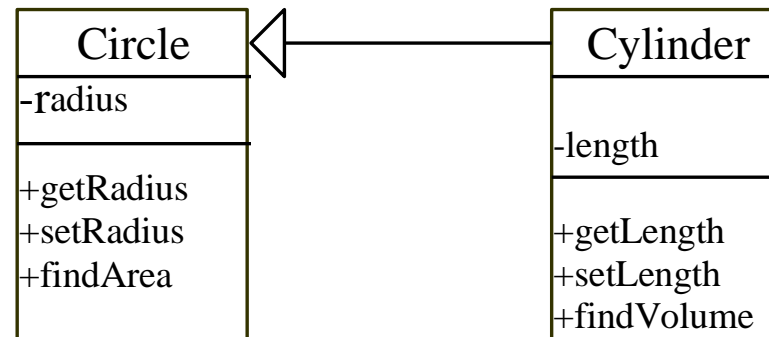
Circle Methods
Cylinder Methods

Circle Data
Cylinder Data

Superclass

Subclass

UML Diagram



Creating a Subclass

11

Creating a subclass extends properties and methods from the superclass. You can also:

- F Add new properties
- F Add new methods
- F Override the methods of the superclass

Overloading and Overriding Methods

12

- **Overloading**: providing more than one method with the **same name but different parameter list**
 - overloading an inherited method means simply adding new method with the same name and different signature

 - **Overriding**: replacing the superclass's implementation of a method with your own design.
 - **both the parameter lists and the return types must be exactly the same**
 - if an overriding method is invoked on an object of the subclass, then it's the **subclass's version** of this method that **gets implemented**
 - an overriding method can have different access specifier from its superclass's version, but only wider accessibility is allowed
- e.g. The **Cylinder** class overrides the **findArea()** method defined in the **Circle** class.

Accessibility and Overriding

13

- A method can be overridden only if it's accessible in the subclass
 - `private` methods in the superclass
 - cannot be overridden
 - if a subclass contains a method which has the same signature as one in its superclass, these methods are totally unrelated
 - `package` methods in the superclass
 - can be overridden if the subclass is in the same package as the superclass
 - `protected, public` methods
 - always will be

Polymorphism

14

- An object of a given class can have multiple forms: either as its declared class type, or as any subclass of it
- an object of an extended class can be used wherever the original class is used

Polymorphism refers to the ability to determine at runtime which code to run, given multiple methods with the same name but different operations in the same class or in different classes. This ability is also known as **dynamic binding**.


- when you invoke a method through an object reference, the *actual class of the object* decides which implementation is used
 - when you access a field, the declared type of the reference decides which implementation is used


Example Classes

15

```
class SuperShow {
    public String str = "SuperStr";
    public void show( ) {
        System.out.println("Super.show:" + str);
    }
}
class ExtendShow extends SuperShow {
    public String str = "ExtendedStr";
    public void show( ) {
        System.out.println("Extend.show:" + str);}

    public static void main (String[] args) {
        ExtendShow ext = new ExtendShow( );
        SuperShow sup = ext;
        sup.show( );          //1
        ext.show( );          //2
        System.out.println("sup.str = " + sup.str);    //3
        System.out.println("ext.str = " + ext.str);    //4
    }
}
```

 **methods invoked through object reference**

 **field access**

Output?

```
Extend.show: ExtendStr
Extend.show: ExtendStr
sup.str = SuperStr
ext.str = ExtendStr
```

Type conversion (1)

16

- The types higher up the type hierarchy are said to be *wider*, or *less specific* than the types lower down the hierarchy. Similarly, lower types are said to be *narrower*, or *more specific*.
- *Widening conversion*: assign a subtype to a supertype
 - ▣ can be checked at compile time. No action needed
- *Narrowing conversion*: convert a reference of a supertype into a reference of a subtype
 - ▣ must be explicitly converted by using the *cast* operator

Type conversion (2)

17

- Explicit type casting: a type name within parentheses, before an expression
 - ▣ for widening conversion: not necessary and it's a safe cast, e.g:

```
String str = "test";  
Object obj1 = (Object)str; ✓  
Object obj2 = str; ✓
```
 - for narrowing cast: must be provided and it's an unsafe cast
e.g.

```
String str1 = "test";  
Object obj = str1;  
String str2 = (String)obj; ✓  
Double num = (Double)obj; ✗
```

E.g. Student is subclass of Person

18

```
public class typeTest {
    static Person[] p = new Person[10];

    static
    {
        for (int i = 0; i < 10; i++) {
            if(i<5)
                p[i] = new Student();
            else
                p[i] = new Person();
        }
    }

    public static void main (String args[]) {
        Person o1 = (Person)p[0];
        Person o2 = p[0];
        Student o3 = p[0]; ✗
        Student o4 = (Student)p[0];
        Student o5 = p[9]; ✗
        Student o6 = (Student)p[9]; ✗
        int x = p[0].getStudentNumber(); ✗
    }
}
```

```
%> javac typeTest.java
```

typeTest.java:17 incompatible types

found : Person

required : Student

Student o3 = p[0];

^

typeTest.java:19 incompatible types

found : Person

required : Student

Student o5 = p[9];

^

typeTest.java:21: cannot resolve symbol

symbol : method getStudentNumber ()

location: class Person

int x = p[0].getStudentNumber()

^

3 errors

After commenting out these three ill lines:

```
%> java typeTest
```

Exception in thread "main"

java.lang.ClassCastException: Person

at typeTest.main(typeTest.java:20)

Type conversion (3)

19

- Type testing: you can test an object's actual class by using the `instanceof` operator

```
e.g. if ( obj instanceof String)
    {
        String str2 = (String)obj;
    }
```

- Use the **instanceof** operator to test whether an object is an instance of a class:

```
Circle myCircle = new Circle();

if (myCircle instanceof Cylinder)
{
    Cylinder myCylinder = (Cylinder)myCircle;
    ...
}
```

Abstract classes and methods (1)

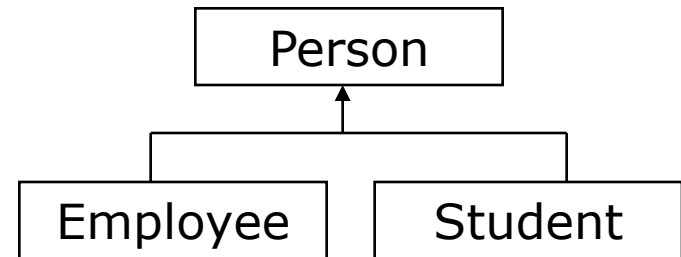
20

- abstract classes: some methods are only declared, but no concrete implementations are provided. They need to be implemented by the extending classes.

```
abstract class Person {  
    String name;  
    . . .  
    public abstract String getDescription();  
    . . .  
}
```

```
Class Student extends Person {  
    private String major;  
    . . .  
    public String getDescription() {  
        return "a student major in " + major;  
    }  
    . . .  
}
```

```
Class Employee extends Person {  
    private float salary;  
    . . .  
    public String getDescription() {  
        return "an employee with a salary of $ " + salary;  
    }  
    . . .  
}
```



Abstract classes and methods (2)

21

- Each method which has no implementation in the **abstract** class must be declared **abstract**
- Any class with any **abstract** methods must be declared **abstract**
- When you extend an `abstract` class, two situations
 1. leave some or all of the abstract methods be still undefined. Then the subclass must be declared as `abstract` as well
 2. define concrete implementation of all the inherited abstract methods. Then the subclass is no longer abstract
- **An object of an abstract class can NOT be created**
 - note that declaring object variables of an abstract class is still allowed, but such a variable can only refer to an object of a non-abstract subclass

E.g. `Person p = new Student();`

The abstract Modifier

22

- The abstract class
 - ▣ Cannot be instantiated
 - ▣ Should be extended and implemented in subclasses
- The abstract method
 - ▣ Method signature without implementation

The `final` Modifier

23

- The `final` class cannot be extended:

```
final class Math  
{...}
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be modified by its subclasses.

Object: the ultimate superclass

24

- The **Object** class is the root of all Java classes: every class in Java extends **Object** without mention
- Utility methods of **Object** class
 - ▣ **equals**: returns whether two object references have the same value
 - ▣ **clone**: returns a clone of the object
 - ▣ **getClass**: return the run expression of the object's class, which is a `Class` object
 - ▣ **toString**: return a string representation of the object

Casting Objects

25

It is always possible to convert a subclass to a superclass. For this reason, explicit casting can be omitted. For example,

```
Circle myCircle = myCylinder
```

is equivalent to:

```
Circle myCircle = (Circle)myCylinder;
```

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Cylinder myCylinder =(Cylinder)myCircle;
```

Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Cylinder myCylinder = (Cylinder)myCircle;
```

Class Design Guidelines

- ❑ Hide private data and private methods.
- ❑ A property that is shared by all the instances of the class should be declared as a class property - **static**.
- ❑ Provide a public default constructor and override the **equals** method and the **toString** method defined in the Object class whenever possible.
- ❑ Choose informative names and follow consistent styles.
- ❑ A class should describe a single entity or a set of similar operations.
- ❑ Group common data fields and operations shared by other classes.

Design hints for inheritance

28

1. Place common operations and fields in the superclass
2. Use inheritance to model a IsA relationship
3. Don't use inheritance unless all inherited methods make sense
4. Don't change the expected behavior when you override a method
5. Use polymorphism, not type information

```
if (x is of type1)
    action1(x);
else if (x is of type2)
    action2(x);
```

Do action1 and action2 represent a common concept? If it is, make the concept a method of a common superclass or interface of both types, and then you can simply call `x.action()`.