Lập trình hướng đối tượng

Biên soạn: Nguyễn Thị Tuyết Hải

Email: tuyethai@ptithcm.edu.vn

Chapter 5: Inheritance

5.1 Classes, Superclasses, and Subclasses

Inheritance: new classes are created relying on existing classes.

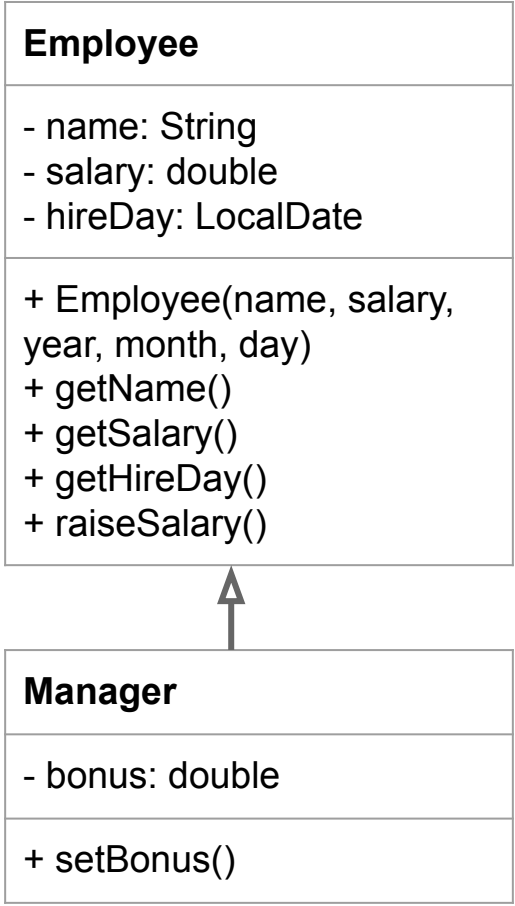
When you inherit from an existing class:

- you reuse its methods
- you can add new methods and fields to adapt your new class to new situations

5.1.1 Defining Subclasses

- extends: a new class derives from an existing class.
  - existing class: superclass, base class, or parent class.
  - new class: subclass, derived class, or child class.
- Subclasses have more functionality than their superclasses.
  - ***Manager*** class encapsulates more data and has more functionality than its superclass ***Employee***

```
public class Manager extends Employee {
    private double bonus;
    ...
    public void setBonus(double bonus) {
        this.bonus = bonus;
    }
}
```

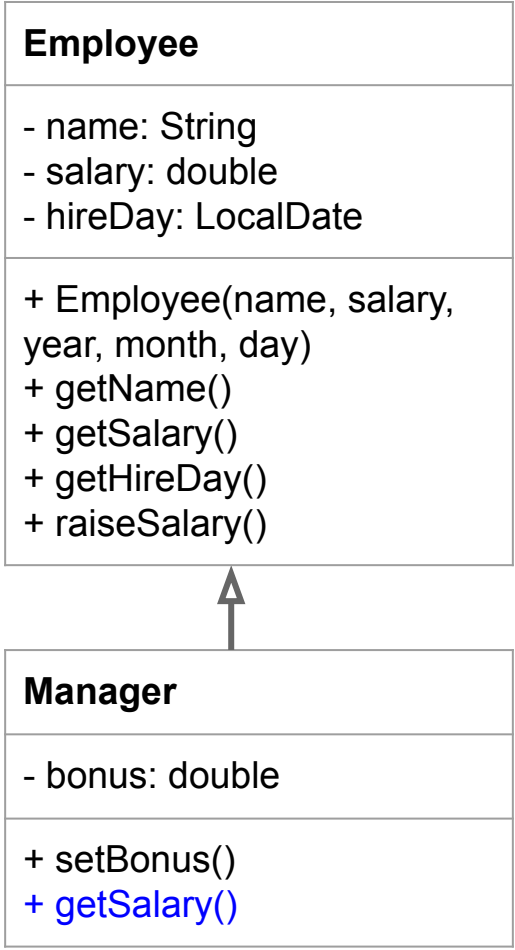


5.1.2 Overriding Methods

- **Override**: a subclass has the same method as declared in the superclass
  - **Manager**: **getSalary()** should return the sum of the base salary and the bonus: ***getSalary() of Employee*** + ***bonus of Manager***

```
public double getSalary() {
    return salary + bonus;
}

public double getSalary() {
    double baseSalary = getSalary();
    return baseSalary + bonus;
}
```

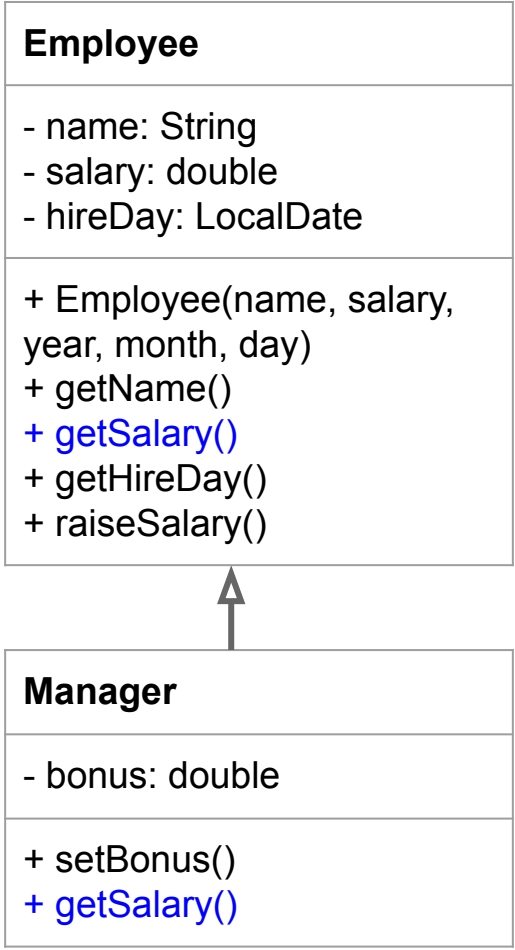


5.1.2 Overriding Methods

- **Override**: a subclass has the same method as declared in the superclass
  - **Manager**: **getSalary()** should return the sum of the base salary and the bonus: ***getSalary() of Employee*** + ***bonus of Manager***

Overloading ? (nap chong)
Overriding ? (ghi de)

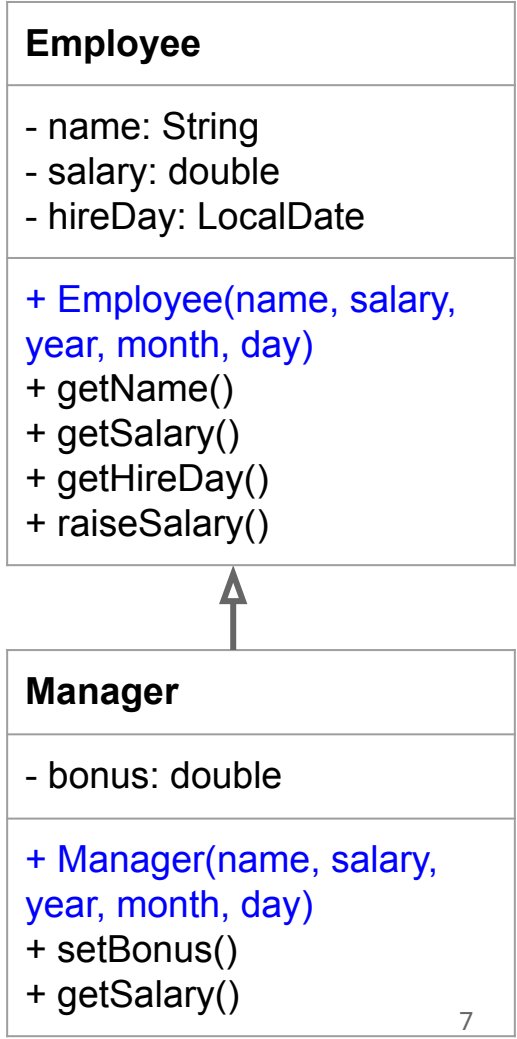
```
public double getSalary() {
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```



5.1.3 Subclass Constructors

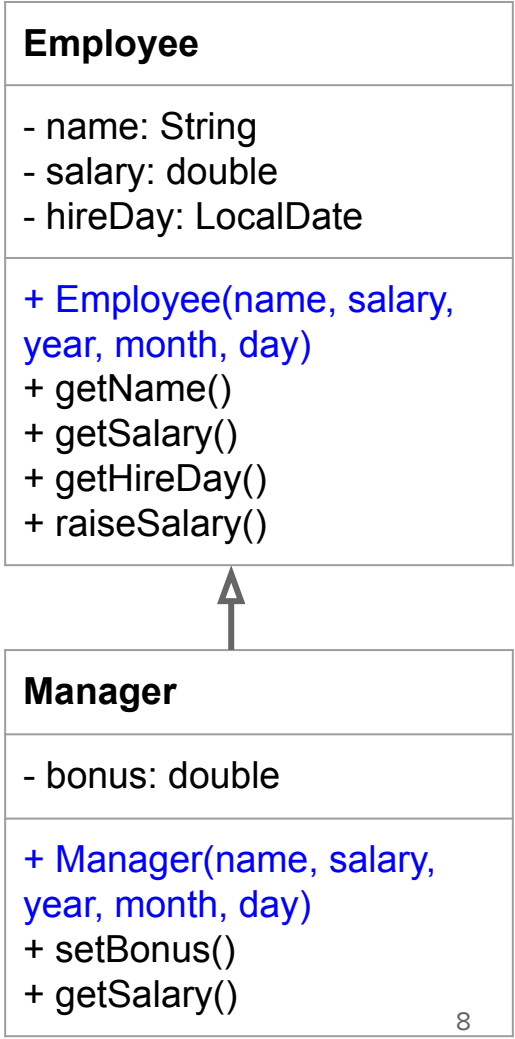
- Add subclass constructors
  - **super** : call the constructor of the superclass
- Subclass constructor call implicitly/explicitly the superclass constructor.
  - Implicitly: the no-argument constructor of the superclass is invoked. If the superclass does not have a no-argument constructor, the Java compiler reports an error.
  - Explicitly: call existing constructor of the superclass by super(...)

```
public Manager(String name, double salary, int year,
               int month, int day) {
    super(name, salary, year, month, day);
    bonus = 0;
}
```



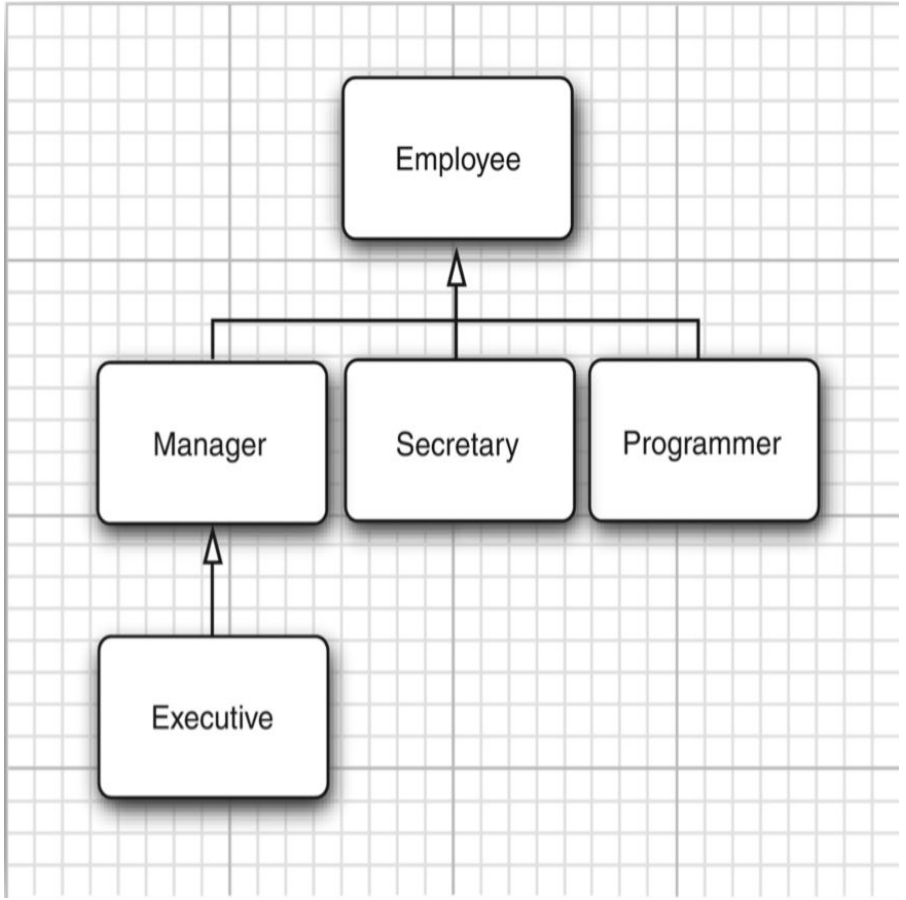
5.1.3 Subclass Constructors

Full code example: **v1ch05.inheritance**



5.1.4 Inheritance Hierarchies

- Inheritance need not stop at deriving one layer of classes.



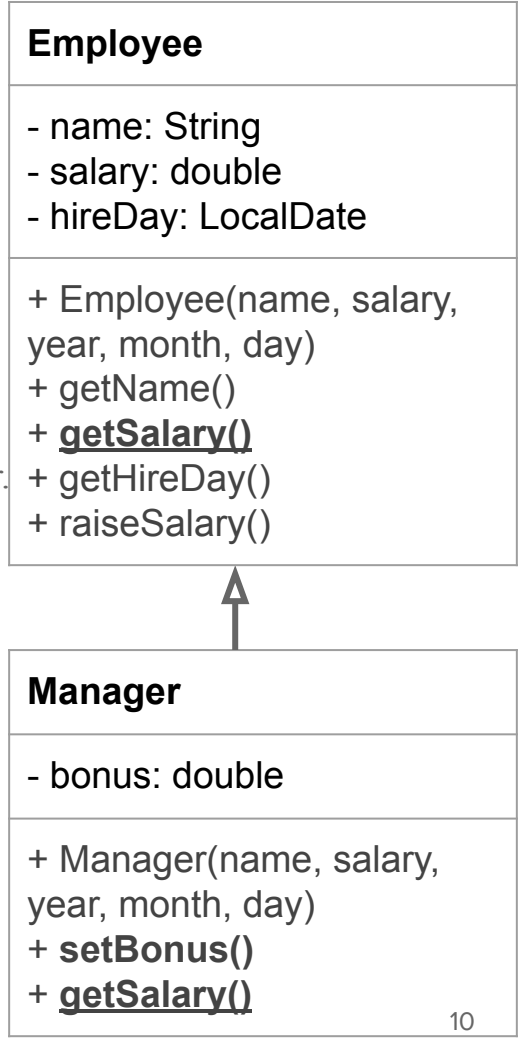
5.1.5 Polymorphism

- The “is–a” rule:
  - Every object of the subclass is an object of the superclass. E.g., every manager is an employee.
  - A **superclass** variable can make a reference to a **subclass** variable.
    - E.g., **staff[0]** and **boss** refer to the same object;
      staff[0] is considered to be only an **Employee** object by the compiler.

```
Manager boss = new Manager(. . .);
Employee[] staff = new Employee[3];

staff[0] = boss; // OK
Manager m = staff[0]; // Error

boss.setBonus(5000); // OK
staff[0].setBonus(5000); // Error
```



5.1.6 Understanding Method Calls

x.f(args),

- x is declared to be an object of **class C**
- list of arguments: **explicit parameters**.

How a method call is applied to an object by the compiler:

1. Looks at the **declared type of the object** and the **method name**.
   a. all methods called **f** in the class **C**
   b. all **accessible** methods called **f** in the **superclasses of C**
2. Determines the **types of the arguments** that are supplied in the method call.
   a. If there is a unique method whose **parameter types** are a best match for the supplied arguments => method is chosen to be called. (**overloading resolution**)
3. Static/dynamic binding:
   a. Static: if the method is **private, static, final, or a constructor**, then the compiler knows which method to call.
   b. Dynamic binding must be used at runtime: depending on **the actual type of the object to which x refers**

VM precomputes for each class a method table that lists all method signatures and the actual methods to be called.

5.1.6 Understanding Method Calls

VM precomputes for each class a method table (all method signatures)
Depending on the **actual type of object**, its method table is looked up and the corresponding method is called.

Employee:

getName() -> Employee.getName()
getSalary() -> Employee.getSalary()
getHireDay() -> Employee.getHireDay()
raiseSalary(double) -> Employee.raiseSalary(double)

Manager:

getName() -> Employee.getName()
getSalary() -> **Manager**.getSalary()
getHireDay() -> Employee.getHireDay()
raiseSalary(double) -> Employee.raiseSalary(double)
setBonus(double) -> **Manager**.setBonus(double)

5.1.7 Preventing Inheritance: Final Classes and Methods

- Prevent someone from inheriting one of your classes: ***final***

If a class is declared **final**, **only the methods,** not the fields, **are automatically final**

```
public final class Executive extends Manager {
...
}
```

- Prevent someone from overriding a specific method in a class***: final***

```
public class Employee {
    ...
    public final String getName() {
        return name;
    }
    ...
}
```

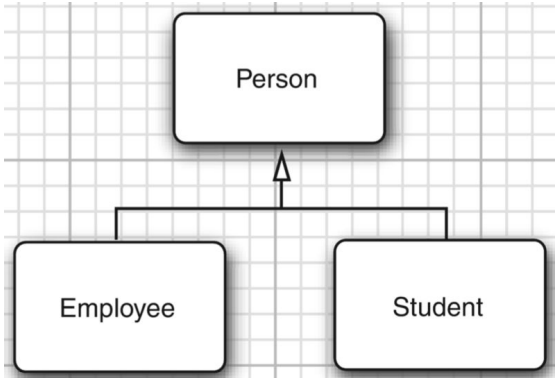
5.1.8 Casting

- Why:
  - to use an object in its full capacity after its actual type has been temporarily forgotten.
- When:
  - only within an inheritance hierarchy.
  - Use ***instanceof*** to check before casting from a superclass to a subclass.

```
public class ManagerTestCast {
        public static void main(String[] args) {
                Employee[] staff = new Employee[3];
                staff[0] = new Manager("Carl Cracker", 80000,
                                                1987, 12, 15);
                staff[0].setBonus(5000); // Error
                // make a cast to use all capacity of Manager
                Manager boss = (Manager) staff[0];
                boss.setBonus(5000);
                // check with instanceof
                if (staff[0] instanceof Manager) {
                        boss = (Manager) staff[0];
                }
        }
}
```

5.1.9 Abstract Classes

- As you **move up** the inheritance hierarchy, classes become **more general**.
- E.g., at a high a level of abstraction: an employee is a person, and a student is, too.
  - getDescription(): return a brief description of the person depending on each subclass.
  - getDescription(): **abstract**
    - no implementation required in an abstract class
    - must implemented in the subclass.




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

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

  public String getName(){
    return name;
  }
}
```

Full code example: v1ch05.abstractClasses

5.1.10 Protected Access

- **protected** modifier: only subclass can access methods/fields of the superclass.
- 4 modifiers:
  - Visible to the class only (**private**).
  - Visible to the world (**public**).
  - Visible to the package and all subclasses (**protected**).
  - Visible to the package—the (unfortunate) default. No modifiers are needed.

5.2 Object: The Cosmic Superclass

Every class in Java extends Object, in default.

```
public class MainTest {
        public static void main(String[] args) {
                Object obj = new Employee("Harry Hacker", 35000, 2000, 10, 1);
                Employee e = (Employee) obj; // OK
        }
}
```

5.2.1 The equals Method

- **equals** method in the Object class: test whether two object references are identical
- state-based equality testing: two objects are equal if they have the same **state**.

```
class Employee {
        public boolean equals(Object otherObject) {
                // a quick test to see if the objects are identical
                if (this == otherObject)
                        return true;
                // must return false if the explicit parameter is null
                if (otherObject == null)
                        return false;
                // if the classes don't match, they can't be equal
                if (getClass() != otherObject.getClass())
                        return false;
                // now we know otherObject is a non-null Employee
                Employee other = (Employee) otherObject;
                // test whether the fields have identical values
                return name.equals(other.name) && salary == other.salary
                                && hireDay.equals(other.hireDay);
        }
}
```

5.2.1 The equals Method

- Define the equals method for a subclass

```
public class Manager extends Employee {
        public boolean equals(Object otherObject) {
                if (!super.equals(otherObject))
                        return false;
                // super.equals checked that this and otherObject belong to the same class
                Manager other = (Manager) otherObject;
                return bonus == other.bonus;
        }
}
```

5.2.2 Equality Testing and Inheritance

```
class Employee {
   public boolean equals(Object otherObject)
   {
      // a quick test to see if the objects are identical
      if (this == otherObject) return true;

      // must return false if the explicit parameter is null
      if (otherObject == null) return false;

      // if the semantics of equals change in subclasses
      if (getClass() != otherObject.getClass()) return false;

      // if the semantics of equals keep the same in subclasses
      if (!(otherObject instanceof Employee)) return false;

      // now we know otherObject is a non-null Employee
      Employee other = (Employee) otherObject;

      // test whether the fields have identical values
      return Objects.equals(name, other.name) && salary == other.salary && Objects.equals(hireDay, other.hireDay);
   }

}
```

5.2.3 The hashCode Method

- The hashCode method is defined in the Object class.
- Every object has a default hashcode that is derived from the object's memory address.

```
String s = "Ok";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());

String t = new String("Ok");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

s: 2556; sb: 2018699554
t: 2556; tb: 1311053135

5.2.4 The toString Method

- Object method: **toString()** returns a string representing the value of this object

Class Employee

```
public String toString() {
        return "Employee[name=" + name + ",salary=" + salary
                        + ",hireDay=" + hireDay + "]";
}
```

```
public String toString() {
        return getClass().getName() + "[name=" + name + ",salary=" + salary
                        + ",hireDay=" + hireDay + "]";
}
```

Class Manager ? toString()

5.2.4 The toString Method

- Object method: toString() returns a string representing the value of this object

Class Manager

```
public String toString() {
        return super.toString() + "[bonus=" + bonus + "]";
}
```

Full codes: v1ch05.equals

5.3 Generic ArrayLists

- Array: set the size of an array at runtime

- ArrayList: automatically adjusts its capacity as we add and remove elements
- ArrayList is a generic class with a type parameter.
- Length: array, arraylist

```
int actualSize = 50;
Employee[] arr_staff = new Employee[actualSize];

ArrayList<Employee> list_staff = new ArrayList<>();

int len = arr_staff.length;
list_staff.size();
```

5.3.1 Accessing ArrayList Elements

- Set the ith element: **set()**
- Get an arraylist element: **get()**
- Add an item: add()
- Remove an item: remove()
- Traverse the contents of an array list

Full code: v1ch05.arrayList;

```
int actualSize = 50;
Employee[] arr_staff = new Employee[actualSize];
ArrayList<Employee> list_staff = new ArrayList<>();

Employee harry = new Employee();
list_staff.set(i, harry);
arr_staff[i] = harry;


Employee tmp1 = list_staff.get(i);
Employee tmp2 = arr_staff[i];

int n = 4;
list_staff.add(n, harry);
Employee tmp3 = list_staff.remove(n);


for (Employee e : list_staff){
        System.out.println(e.toString());
}

for (i = 0; i < list_staff.size(); i++) {
        Employee etmp = list_staff.get(i);
        System.out.println(etmp.toString());
}
```

5.4 Object Wrappers and Autoboxing

- We want an arraylist of integers: ~~ArrayList<int> list~~
- All primitive types have class counterparts (wrappers), e.g., **Integer** corresponds to **int**
- 6 wrappers: Integer, Long, Float, Double, Short, Byte, Character, and Boolean.
  - Immutable
  - Final
  - Inherit from Number.
  - E.g., ArrayList<**Integer**> list
- Automatic boxing and unboxing work automatically: wrappers ⇔ counterparts
  - E.g.
  - `list.add(3); // list.add(Integer.valueOf(3))`
  - `int n = list.get(i); // int n = list.get(i).intValue();`
  - `Integer n = 3; n++;`
- To convert a string to an integer:
  - `int x = Integer.parseInt(s);`

```
Integer a = 1000; Integer b = 1000; a==b?
```

5.6 Enumeration Classes

- `public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };`
- Enum class with additional constructors, accessor

```
enum Size {
        SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
        private String abbreviation;

        private Size(String abbreviation) {
                this.abbreviation = abbreviation;
        }

        public String getAbbreviation() {
                return abbreviation;
        }
}
Size size = Enum.valueOf(Size.class, "SMALL");
Size[] values = Size.values();
```

Full codes: v1ch05.enums

5.6 Enumeration Classes

```
enum Size {
        SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
        private String abbreviation;

        private Size(String abbreviation) {
                this.abbreviation = abbreviation;
        }

        public String getAbbreviation() {
                return abbreviation;
        }
}
Size size = Enum.valueOf(Size.class, "SMALL");
Size[] values = Size.values();
```

Full codes: v1ch05.enums

5.7 Reflection

A program that can analyze the capabilities of classes is called reflective
It is of interest mainly to tool builders, not application programmers

5.7.1 The Class class

- Java runtime system always maintains **runtime type identification** (the class to which each object belongs) on all objects
- the class that holds this information is called, **Class**
- **getClass()** method in the Object class returns an instance of Class type.

```
Employee e;
...
Class cl = e.getClass();
System.out.println(e.getClass().getName() + " " + e.getName());
```

- obtain a Class object corresponding to a class name by using the static forName method.

```
String className = "java.util.Random";
Class cl = Class.forName(className);
```

5.8 Design Hints for Inheritance

1. Place common operations and fields in the superclass.

2. Don't use **protected** fields.
   - protected mechanism doesn't give much protection
     - anyone can form a **subclass** of **your classe**s and write code that directly accesses protected instance fields
     - **all classes in the same package** have access to protected fields

   - protected methods can be useful to indicate methods that are **not ready for general use** and **should be redefined in subclasses**.

3. Use inheritance to model the "is–a" relationship.

4. Don't use inheritance unless all inherited methods make sense.

```
class Holiday extends GregorianCalendar { . . . }
Holiday christmas;

// And add() can turn holidays into non-holidays:
christmas.add(Calendar.DAY_OF_MONTH, 12);
```

5.8 Design Hints for Inheritance

5. Don't change the expected behavior when you **override** a method.
E.g., you can "fix" the issue of the **add()** in the Holiday class by redefining **add()**, perhaps to do nothing, … -> a fix violates inheritance rules.

6. Use polymorphism, not type information.

- Multiple type tests

```
if(x is of type 1)
        action1(x);
elseif(x is of type 2)
        action2(x);
```

- Do action1 and action2 **represent a common concept**? If yes, think polymorphism **x.action()**;