Програмування-1

Лекція 9

Основи ООП

План лекції

- Модифікатори доступу та області видимості
- **final-** класи, методи, поля, змінні, параметри
- **enum** – Перелічуваний тип
- Абстрактні класи
- Інтерфейси

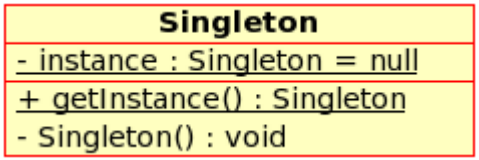
Модифікатори доступу та області видимості

- 3 модифікатора
  - **private**
  - **protected**
  - **public**
- 4 області видимості

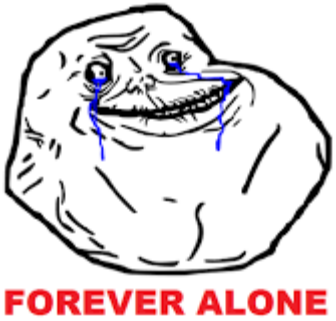
Область видимості	Той же клас	Той же пакет	Клас-нащадок	Будь хто
private	Так	-	-	-
- *(default)*	Так	Так	-	-
protected	Так	Так	Так	-
public	Так	Так	Так	Так

Модифікатори доступу та конструктори

- Конструктор за замовчуванням має таку саму область видимості що й сам клас
- Якщо не можна побачити конструктор – неможна створити об’єкт за допомогою оператора **new**
  - застосування:
    - утилітні класи (Math, System)
    - патерн Singleton




```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```



Модификаторы доступу та успадкування

- При заміщенні (**override**) метода, область видимості в підкласі:
  - можна залишити таку саму
  - можна розширити (ослабити)
  - неможна звужувати

private-методи:

неможливо побачити =
неможливо замістити

- Причина: клас-нащадок розширює базовий клас, тобто повинен мати усі можливості свого предка
  - LSP - Liskov substitution principle

final

- **final клас** – неможливо розширити (**no extends**)
  - приклад: клас **String**
- **final метод** – неможна замістити при наслідуванні (**no override**)
  - у final-класі, все методи неявно є final
  - private методи неявно є final
- **final поле** – після ініціалізації неможливо змінити (**no change**)
  - immutable класи
  - константи
    - в Java слово **const** ніяк не використовується
    - до Java 5 усі константи створювали як **public static final**
    - у Java 5 створили **enum** (далі буде)
- **final параметри та локальні змінні**
  - компілятор гарантує відсутність змін
  - підвищення читабельності
  - зниження людського фактору
  - **замикання** (**closure**): доступ до локальних змінних з локальних та анонімних класів (далі буде)
    - починаючи з Java 8 з'явився термін **effectively final**

Ініціалізація final поля

- Для **final** поля немає значення за замовченням
  - потрібна **явна** ініціалізація
- Компілятор гарантує відсутність змін або повторної ініціалізації
  - потрібна одна і лише одна явна ініціалізація

static поля	non-static поля
- при декларації поля - або в статичному блоці	- при декларації поля - або в нестатичному блоці ініціалізації - або в конструкторах

```
class StaticFinalDemo {

    // Init here
    final static int finalStatic = 1;

    static {
        // OR here
        // finalStatic = 2;
    }
}
```

```
class NonStaticFinalDemo {

    // Init here
    final int finalNonStatic = 1;

    {
        // OR here
        // finalNonStatic = 2;
    }

    NonStaticFinalDemo() {
        // OR here
        // finalNonStatic = 3;
    }
}
```

пам'ятаємо про ***this(...)***

enum

- Раніше для констант використовували **int**
- Недоліки **int** для номінальних та порядкових шкал:
  - можна помилково вказати константу із іншого типу
  - можна помилково вказати невірне число замість константи
- У **Java 5** з'явився **enum**
  - компілятор відслідковує сумісність типів
  - **enum** це клас, тому може мати поля, конструктори та методи
  - **enum** може реалізовувати інтерфейси
  - **enum** не може розширювати класи
    - будь-який enum неявно розширює java.lang.Enum
  - можна використовувати з оператором **switch-case-default**
  - гарантується, що буде створено не більше одного екземпляра для кожної константи
    - використовується для реалізації патерна Singleton
    - для порівняння можна використовувати «**==**» замість «**equals()**»
  - дивись http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

final VS enum

public static final	enum

`class Month {`
`  public static final int JANUARY=1;`
`  public static final int FEBRUARY=2;`
`  public static final int MART=3;`
`}`
 	
`public class FinalExample {`
 	
`  void setDate(int day, int month) {`
`  // ...`
`  }`
 	
`  void someMethod() {`
 	
`  // Compiled & Ok`
`  setDate(14, Month.FEBRUARY);`
 	
`  // Error, but compiled ☹`
`  setDate(Month.MART, 8);`
`  }`
`}`
	`enum Month {`
	`  JANUARY, FEBRUARY, MART;`
	`}`
	
	`public class EnumExample {`
	
	`  void setDate(int day, Month month) {`
	`  // ...`
	`  }`
	
	`  void someMethod() {`
	
	`  // Compiled & Ok`
	`  setDate(14, Month.FEBRUARY);`
	
	`  // Error and NOT compiled ☺`
	`  setDate(Month.MART, 8);`
	`  }`
	`}`

Абстрактні класи

- Абстрактний клас – «Клас-заготовка» на основі якої за допомогою успадкування пізніше будуть створені конкретні класи-нащадки
  - відношення IS-A
  - поліморфізм
- При декларації класу використовують модифікатор **abstract**
- Може містити декларації методів
  - модифікатор **abstract**
  - крапка з комою «**;**» замість тіла метода « **{ }** »
- Якщо є хоча б один абстрактний метод –> маємо абстрактний клас
  - клас може бути абстрактним навіть якщо в ньому немає абстрактних методів
- Неможливо створити об’єкт абстрактного класу
  - можна використовувати посилання абстрактного типу для доступу до об’єктів конкретних класів
- Можуть мати конструктори
  - ці конструктори є ланками у ланцюгу конструкторів від конкретного класу до Object

Абстрактні класи – приклади

- Calendar
  - GregorianCalendar
- InputStream
  - AudioInputStream, ByteArrayInputStream, FileInputStream, FilterInputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream
- OutputStream
  - ByteArrayOutputStream, FileOutputStream, FilterOutputStream, ObjectOutputStream, PipedOutputStream
- ClassLoader
  - SecureClassLoader
- TimerTask
  - MyHelloWorldTimerTask ☺

Абстрактні класи – приклад

```
abstract class Person {

  String name;

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

  abstract public void sayHello();

  public void doHandshake(Person p) {
    // do handshake with p
    System.out.println(
      "----- " + name +
      " give hand to " + p.name +
      " -----");
  }

  public void greet(Person p) {
    doHandshake(p);
    sayHello();
  }

}
```

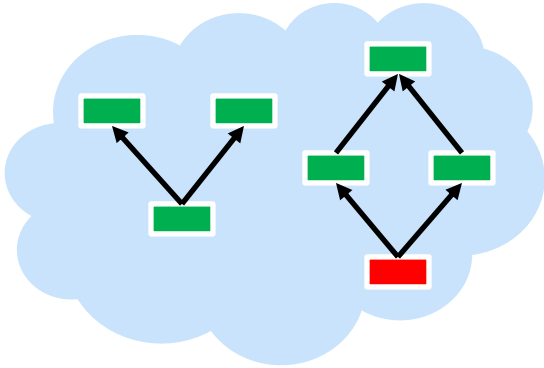
```
class UkrainianPerson extends Person {

  public UkrainianPerson(String name) {
    super(name);
  }

  public void sayHello() {
    System.out.println("Вітаю!");
  }
}


class AmericanPerson extends Person {

  public AmericanPerson(String name) {
    super(name);
  }

  public void sayHello() {
    System.out.println("Hello!");
  }

}
```

Абстрактні класи – приклад (2/2)

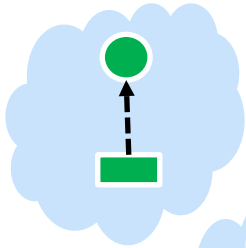
```
public class Test {

   public static void main(String[] args) {

      Person ivan = new UkrainianPerson("Іван");
      Person john = new AmericanPerson("John");

      ivan.greet(john);
      john.greet(ivan);

         }
}
```

```
----- Іван give hand to John -----
Вітаю!
----- John give hand to Іван -----
Hello!
```

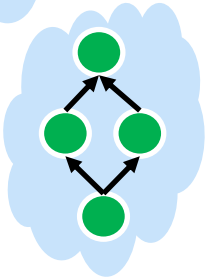
Інтерфейси



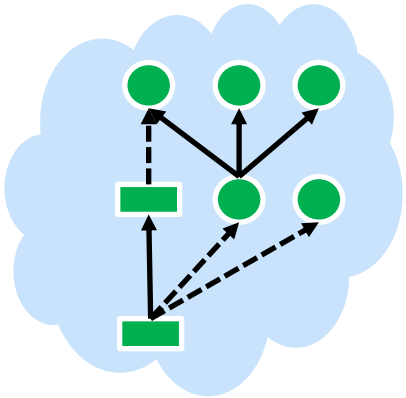
- В Java немає множинного наслідування класів
  - немає проблеми ромба (diamond problem)
  - клас має лише одного безпосереднього предка
  - для того, щоб показати що клас IS-A «щось» і при цьому IS-A «ще щось», використовуються интерфейси



- Інтерфейс – абстрактний тип, містить лише декларації методів та константи
  - контракт про те, які методи мають бути присутні у класі, що реалізовує даний інтерфейс



- Інтерфейс може розширювати інший/інші інтерфейси
  - для інтерфейсів є множинне наслідування
  - оскільки інтерфейси містять лише декларації методів (без реалізації) немає проблеми ромба
- Клас може реалізовувати декілька інтерфейсів
  - оскільки інтерфейси містять лише декларації методів (без їх реалізації) немає проблеми ромба



Інтерфейси – синтаксис

- Синтаксис:

```
<modifier> interface <Name> [extends <interface> [,<interface>]* ] {
    <member_declaration>*
}
```

- В інтерфейсах (Java 1 – Java 7) можуть бути присутні:
  - **декларації методів**
    - **завжди public abstract**, навіть якщо це не указано явно
    - не можуть бути **private**, **protected**, **final** **static**
  - **константи**
    - **завжди public static final**, навіть якщо це не указано явно
    - не можуть бути **private**, **protected**
- В Java 8 також можна додавати:
  - **static** методи
  - реалізацію за-замовчуванням (**default**)
- В Java 9 також можна додавати:
  - **private** методи

Інтерфейси – приклад 1

```
interface Speakable {
   void speak(); // implicitly public abstract
}

class Human implements Speakable {
   public void speak() { // must be public
      System.out.println("Hello! How are You?");
   }
}

class Cat implements Speakable {
   public void speak() { // must be public
      System.out.println("Meow");
   }
}

class InterfaceDemo {

   public static void main(String[] args) {

      Speakable[] creatures = {new Human(), new Cat(), new Cat()};

      for (Speakable s:creatures) {
         s.speak();
      }
   }
}
```

Интерфейсы – пример 2

```
interface Animal {
    void eat(); // eat something
    void sleep(); // sleep somewhere
    void walk(); // walk
    void run(); // walk very fast
}

// java.lang:
// interface Runnable {
//   abstract public void run(); // execute code in parallel thread
// }

class Cat implements Animal, Runnable {
    public void eat() {
        // eat whiskas, drink milk
    }

    public void sleep() {
        // sleep in warm place
    }

    public void walk() {
        // walk quietly
    }

    public void run() {
        // ??????????????????????
    }
}
```

Ретельно обирайте назви методів у інтерфейсах.

Намагайтесь уникати конфліктів імен з методами інших інтерфейсів

В Java неможна реалізувати в одному класі різні методи з однаковою сигнатурою із різних інтерфейсів

«Нестандартне» використання інтерфейсів

- Антіпатерн «Сховище констант»
  - http://en.wikipedia.org/wiki/Constant_interface

```
interface Constants {
    double PI = 3.14159;
    double PLANCK_CONSTANT = 6.62606896e-34;
}

class Calculations implements Constants {
    public double getReducedPlanckConstant() {
        return PLANCK_CONSTANT / (2 * PI);
    }
}
```

- Патерн «Маркерний інтерфейс»

java.io.Serializable:

```
public interface Serializable {
}
```

```
import java.io.Serializable;

class MyClass {
    void myMethod(Object o) {
        if (o instanceof Serializable) {
            // do something
        }
    }
}
```

Інтерфейси vs Абстрактні класи

Можливість	Абстрактний клас	Інтерфейс
Множинне наслідування/реалізація	-	+
Декларація абстрактних методів	+	+
Реалізація конкретних методів	+	-
Декларація констант	+	+
Декларація полів	+	-
Конструктори	+	-
Створення екземплярів	-	-
Область видимості елементів	любая	public