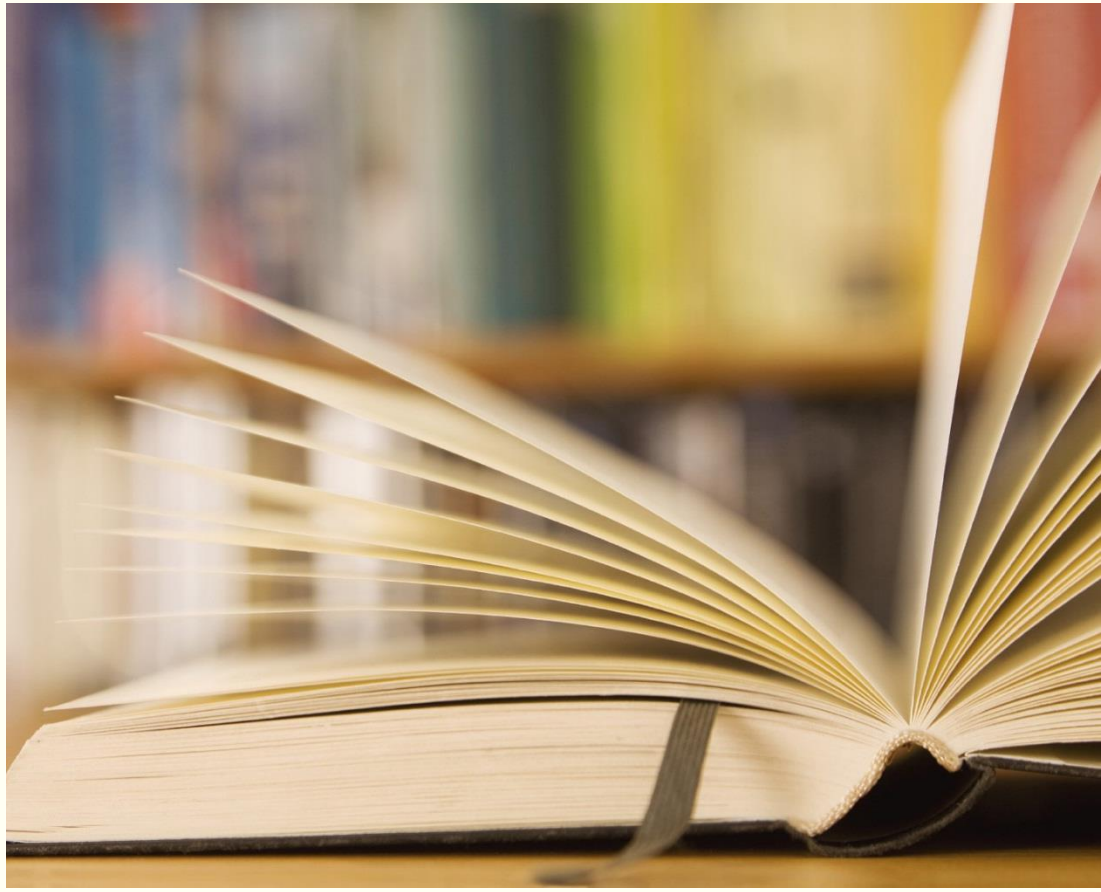
ПОЛІМОРФІЗМ, НАСЛІДУВАННЯ ТА УЗАГАЛЬНЕНЕ ПРОГРАМУВАННЯ МОВОЮ JAVA

Лекція 03
Java-програмування



План лекції

- Поліморфізм. Зміна форми
- Інтерфейси та їх реалізація
- Узагальнені типи (дженерики) Java
- Лямбда-вирази та функціональні інтерфейси

ПОЛІМОРФІЗМ. ЗМІНА ФОРМИ

Питання 3.1.

Зміна форми

- Деякі реальні об’єкти можуть змінювати форму: вода стає твердою, рідкою або газоподібною.
- Поліморфізм – здатність змінювати форму.
  - Код, який рисує поширені геометричні фігури, краще всього організувати на базі одного класу Shape та одного методу draw().
  - Виклик методу для екземпляру Circle нарисує коло, для Rectangle – прямокутник тощо. Існує багато форм методу draw(), він поліморфний.
- Java підтримує 4 види поліморфізму:
  - *Coercion (зведення типу, примусовий поліморфізм)*
  - *Overloading (перевантаження)*
  - *Parametric (параметричний поліморфізм)*
  - *Subtype (поліморфізм підтипів)*

Види поліморфізму

- *Coercion:* ***неявне зведення типів.***
  - Приклад: операція ділення цілих та дробових чисел.
  - Інший приклад: передача посилання на об'єкт підкласу в параметр методу суперкласу. Компілятор неявно зводить тип підкласу до типу суперкласу з метою обмежити операції лише доступними з суперкласу.
- ***Перевантаження:*** один символ оператора або назва методу може використовуватись у різних контекстах.
  - Приклад: «+» використовується для додавання цілих чисел, дробових чисел, конкатенації рядків залежно від типів операндів.
  - Багато методів з однією назвою може з'являтись у класі (шляхом оголошення та/або наслідування).
- Багато програмістів не вважають перевантаження та неявне зведення типів видами поліморфізму.

Види поліморфізму

- ***Параметричний поліморфізм:***
  - В оголошенні класу назва поля може пов'язуватись з різними типами, а назва методу – з різними параметрами та типами повернення (return types).
  - Поле та метод далі можуть набувати різних типів у кожному екземплярі класу.
    - Наприклад, поле може бути типу java.lang.Integer і метод може повертати Integer в одному екземплярі класу, а в іншому екземплярі класу поле і дані, що повертаються від методу, можуть бути типу String.
    - Java підтримує параметричний поліморфізм за допомогою дженериків.
- ***Поліморфізм підтипів:***
  - Тип може слугувати підтипом для іншого типу.
  - Коли екземпляр підтипу з'являється в контексті супертипу, виконання операції з супертипом на екземплярі підтипу призводить до виконання версії операції для підтипу.
  - Наприклад, нехай клас Circle є підкласом Point, і обидва класи мають метод draw().
    - Присвоєння екземпляру Circle змінній типу Point та виклик методу draw() для цієї змінної призводить до виклику версії методу з класу Circle.

Поліморфізм підтипів. Зведення до базового типу

```
class Circle extends Point
{
   private int radius;

   Circle(int x, int y, int radius)
   {
      super(x, y);
      this.radius = radius;
   }

   int getRadius()
   {
      return radius;
   }

   @Override
   public String toString()
   {
      return "" + radius;
   }
}
```

- Ви можете розширити клас Point класом Circle, який представляє поле для радіуса.
  - Клас Circle передбачає, що можна звертатись до екземпляру Circle начебто це екземпляр Point.

```
Circle c = new Circle(10, 20, 30);
Point p = c;
```

- Зведення типів непотрібне, оскільки доступ до екземпляру класу Circle за допомогою інтерфейсу Point легальний.
  - Таке присвоєння називають зведенням до базового типу (*upcasting*), оскільки Ви неявно піднімаєтесь по ієрархії типів.
- Після зведення Circle до базового класу Point неможливо викликати метод getRadius() класу Circle, він не є частиною інтерфейсу Point.
  - Втрата доступу до функціоналу підтипів після звуження підкласу до суперкласу здається недоцільною, проте вона необхідна для досягнення поліморфізму підтипів
- Також поліморфізм підтипів включає оголошення методу в суперкласі та його переозначення в підкласі.
  - Наприклад, Point та Circle є частиною графічного додатку, і вам потрібно представити метод draw() в кожному класі, щоб нарисувати точку та коло відповідно.

```
class Point
{
   private int x, y;

   Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   int getX()
   {
      return x;
   }

   int getY()
   {
      return y;
   }

   @Override
   public String toString()
   {
      return "(" + x + ", " + y + ")";
   }

   void draw()
   {
      System.out.println("Point drawn at " + toString());
   }
}
```

```
class Circle extends Point
{
   private int radius;

   Circle(int x, int y, int radius)
   {
      super(x, y);
      this.radius = radius;
   }

   int getRadius()
   {
      return radius;
   }

   @Override
   public String toString()
   {
      return "" + radius;
   }

   @Override
   void draw()
   {
      System.out.println("Circle drawn at " + super.toString() +
                         " with radius " + toString());
   }
}
```

Тестування написаних класів

```
public class Graphics
{
   public static void main(String[] args)
   {
      Point[] points = new Point[] { new Point(10, 20), new Circle(10, 20, 30) };
      for (int i = 0; i < points.length; i++)
         points[i].draw();
   }
}
```

- Метод main() спочатку оголошує масив даних типу Point.
  - Зведення до базового типу (Upcasting) демонструється наявністю інстанціювання ініціалізатора масиву класу Circle.
  - Далі main() використовує цикл for для виклику методу draw() для кожного елементу Point.
- Вивід:

```
Point drawn at (10, 20)
Circle drawn at (10, 20) with radius 30
```

Раннє та пізнє зв’язування

- Як Java «знає», що потрібно викликати метод draw() класу Circle на другій ітерації циклу?
  - Може не викликати метод draw() класу Point, оскільки Circle обробляється так же, як і Point завдяки зведенню до базового типу?
- Під час компіляції невідомо, який метод викликати.
  - Єдине, що можна, - перевірити існування методу в суперкласі та переконатись, що список аргументів викликаного методу та return type співпадають з оголошенням методу суперкласу.
- Тому компілятор вставляє інструкцію у скомпільований код, яка під час виконання отримує та використовує посилання
  - Таку задачу називають ***пізнім зв’язуванням***.
  - Використовується для викликів неконстантних методів екземпляру.
  - Для інших методів компілятор знає, який метод викликати, і вставляє інструкцію у скомпільований код, який викликає метод, пов’язаний з типом змінної (а не значенням).
  - Така задача відома як ***раннє зв’язування (early binding)***.

Демонстрація зведення масиву до базового типу (Array Upcasting)

```
class Point
{
   private int x, y;

   Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   int getX() { return x; }
   int getY() { return y; }
}
```

```
class ColoredPoint extends Point
{
   private int color;

   ColoredPoint(int x, int y, int color)
   {
      super(x, y);
      this.color = color;
   }

   int getColor() { return color; }
}
```

```
public class UpcastArrayDemo
{
   public static void main(String[] args)
   {
      ColoredPoint[] cptArray = new ColoredPoint[1];
      cptArray[0] = new ColoredPoint(10, 20, 5);
      Point[] ptArray = cptArray;
      System.out.println(ptArray[0].getX()); // Output: 10
      System.out.println(ptArray[0].getY()); // Output: 20
//      System.out.println(ptArray[0].getColor()); // Illegal
   }
}
```

Абстрактні класи та абстрактні методи

- Нехай нові вимоги диктують потребу включення класу Rectangle у графічний додаток.
  - Крім того, клас має містити метод draw(), який необхідно тестувати.
- На відміну від класу Circle (точка з радіусом), немає сенсу вважати Rectangle як точку (Point) з висотою та шириною.
  - Екземпляр класу Rectangle, скоріше, буде скомпонований з 2 точок (екземплярів Point).
  - Оскільки кола, точки та прямокутники є прикладами фігур, більш осмисленим є оголошення класу Shape з власним методом draw(), ніж задавати клас Rectangle extends Point.

```
class Shape
{
   void draw()
   {
   }
}
```

Рефакторинг класу Graphics

```
public class Graphics
{
   public static void main(String[] args)
   {
      Shape[] shapes = new Shape[] { new Point(10, 20), new Circle(10, 20, 30),
                                     new Rectangle(20, 30, 15, 25) };
      for (int i = 0; i < shapes.length; i++)
         shapes[i].draw();
   }
}
```

- Оскільки класи Point та Rectangle напряму, а клас Circle опосередковано (через Point), розширяють клас Shape, метод main() викличе підходящий метод draw() з підкласу у відповідь на вираз shapes[i].draw();.

Код гнучкіший, проте існує проблема

- Що зупинить розробника від додавання такого беззмістовного екземпляру в масив?

```
Shape[] shapes = new Shape[] { new Point(10, 20), new Circle(10, 20, 30),
                               new Rectangle(20, 30, 15, 25), new Shape() };
```

- Оскільки клас Shape описує абстрактне поняття, що значить нарисувати generic shape?
- Вирішення:

```
abstract class Shape
{
   abstract void draw();
}
```

- Передбачено зарезервоване слово abstract для створення класів, що не можуть бути інстанційованими.
  - Компілятор видає помилку при спробі інстанціювання цього класу.

Абстрактні та фінальні класи та методи

- Також зарезервоване слово abstract використовується для оголошення методу без тіла
  - Метод draw() не потребує тіла, оскільки неможливо нарисувати абстрактну фігуру.
- Помилка: при спробі оголосити клас як abstract і final.
  - Абстрактний клас не можна інстанціювати, а константний клас неможливо розширити.
- Помилка: оголошується абстрактний метод в неабстрактному класі.
  - Неабстрактний (конкретний) клас не може бути інстанційованим, якщо містить абстрактний метод.
- При розширенні абстрактного класу, у розширеному класі необхідно переозначити (override) всі абстрактні методи абстрактного класу.
  - Інакше розширений клас сам має бути абстрактним; якщо ні – компілятор видасть помилку.

Понижуюче зведення (downcasting)

- Підйом по ієрархії за допомогою upcasting спричиняє втрату доступу до можливостей підтипів.
  - Point p = new Circle(r); означає, що Ви не можете використовувати p для виклику методу getRadius() класу Circle.
  - Проте можливо відновити доступ до методу getRadius(), виконуючи явне зведення типу, наприклад, Circle c = (Circle) p;.
  - Таке присвоєння називається понижуючим зведенням (downcasting), оскільки Ви явно спускаєтесь по ієрархії типів (від суперкласу Point до субкласу Circle).
  - Хоч зведення до базового типу завжди безпечне, про понижуюче зведення таке сказати не можна.

Проблема з понижуючим зведенням

```
class A
{
}

class B extends A
{
   void m()
   {
   }
}

public class DowncastDemo
{
    public static void main(String[] args)
    {
       A a = new A();
       B b = (B) a;
       b.m();
    }
 }
```

- Ієрархія класів складається із суперкласу A та підкласу B.
  - Клас B оголошує єдиний метод m().
- Третій клас – DowncastDemo – постачає метод main(), який спочатку інстанціює A, а потім намагається понизити тип екземпляру до B та присвоїти результат змінній b.
  - Код легальний, компілятор не знайде помилки.
  - Проте застосунок буде «падати», коли дійде черга до виконання коду b.m();.
    - Віртуальна машина намагатиметься викликати метод, який не існує.
    - Клас A не має методу m().

Такий сценарій ніколи не трапиться

- Віртуальна машина перевіряє зведення на легальність до виконання операції зведення.
  - Оскільки A не має методу m(), ВМ не дозволить зведення, викинувши виключення типу ClassCastException.
  - Перевірка зведення типів віртуальною машиною ілюструє ***динамічну ідентифікацію типів даних*** (*runtime type identification* (*RTTI*)).
    - Відбувається перегляд типу операнду оператора зведення на можливість цієї дії.
  - Друга форма RTTI включає оператор instanceof, який перевіряє лівий операнд на те, чи є він екземпляром правого операнду.
    - Якщо так, повертає true.
    - Для виключення появи ClassCastException:

```
if(a instanceof B)
{
    B b = (B) a;
    b.m();
}
```

Оператор instanceof

- Оскільки підтип є певною мірою супертипом, оператор instanceof поверне true, коли його лівий операнд є екземпляром підтипу або супертипу правого операнду:

```
A a = new A();
B b = new B();
System.out.println(b instanceof A); // Output: true
System.out.println(a instanceof A); // Output: true
```

- У прикладі структура класів відповідає структурі з попередніх слайдів та інстанціює суперклас A та підклас B.
  - Перший виклик System.out.println() виводить true, оскільки посилання b identifies екземпляр підкласу класу A;
  - Другий виклик System.out.println() виводить true, оскільки посилання a identifies екземпляр суперкласу A.

```
class Point
{
   private int x, y;

   Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   int getX() { return x; }
   int getY() { return y; }
}

class ColoredPoint extends Point
{
   private int color;

   ColoredPoint(int x, int y, int color)
   {
      super(x, y);
      this.color = color;
   }

   int getColor() { return color; }
}
```

Демонстрація Array Downcasting

- Оператор instanceof використовується для перевірки того, що об'єкт, на який посилається ptArray's, має тип ColoredPoint[].
  - Якщо оператор поверне true, понижуюче зведення ptArray[0] від Point до ColoredPoint безпечне, як і присвоєння посилання to ColoredPoint.

```
public class DowncastArrayDemo
{
   public static void main(String[] args)
   {
      ColoredPoint[] cptArray = new ColoredPoint[1];
      cptArray[0] = new ColoredPoint(10, 20, 5);
      Point[] ptArray = cptArray;
      System.out.println(ptArray[0].getX()); // Output: 10
      System.out.println(ptArray[0].getY()); // Output: 20
//      System.out.println(ptArray[0].getColor()); // Illegal
      if (ptArray instanceof ColoredPoint[])
      {
         ColoredPoint cp = (ColoredPoint) ptArray[0];
         System.out.println(cp.getColor());
      }
   }
}
```

Коваріантні типи повернення (Covariant Return Types)

- Типи повернення методу, які (в оголошенні методу в суперкласі) є супертипом для типу повернення в переозначеному оголошенні методу підкласу.

```
class SuperReturnType
{
   @Override
   public String toString()
   {
      return "superclass return type";
   }
}

class SubReturnType extends SuperReturnType
{
   @Override
   public String toString()
   {
      return "subclass return type";
   }
}
```

```
class Superclass
{
   SuperReturnType createReturnType()
   {
      return new SuperReturnType();
   }
}

class Subclass extends Superclass
{
   @Override
   SubReturnType createReturnType()
   {
      return new SubReturnType();
   }
}
```

https://habr.com/ru/post/218753/

```
public class CovarDemo
{
    public static void main(String[] args)
    {
        SuperReturnType suprt = new Superclass().createReturnType();
        System.out.println(suprt); // Output: superclass return type
        SubReturnType subrt = new Subclass().createReturnType();
        System.out.println(subrt); // Output: subclass return type
    }
}
```

- Коваріантні типи повернення мінімізують upcasting та downcasting.
  - Наприклад, метод createReturnType() підкласу Subclass не потребує зведення екземпляру класу SubReturnType до базового типу повернення SubReturnType.
  - Крім того, цей екземпляр не вимагає понижуючого зведення (downcasting) до типу SubReturnType, коли відбувається присвоєння змінній subrt.

Без коваріантних типів повернення

```
class SuperReturnType
{
   @Override
   public String toString()
   {
      return "superclass return type";
   }
}

class SubReturnType extends SuperReturnType
{
   @Override
   public String toString()
   {
      return "subclass return type";
   }
}

class Superclass
{
   SuperReturnType createReturnType()
   {
      return new SuperReturnType();
   }
}
```

```
class Subclass extends Superclass
{
   @Override
   SuperReturnType createReturnType()
   {
      return new SubReturnType();
   }
}

public class CovarDemo
{
   public static void main(String[] args)
   {
      SuperReturnType suprt = new Superclass().createReturnType();
      System.out.println(suprt); // Output: superclass return type
      SubReturnType subrt = (SubReturnType) new Subclass().createReturnType();
      System.out.println(subrt); // Output: subclass return type
   }
}
```

ДЯКУЮ ЗА УВАГУ!

Наступне запитання: інтерфейси та їх реалізація