ІНТЕРФЕЙСИ ТА ЇХ РЕАЛІЗАЦІЯ

Питання 3.2.

Формалізація інтерфейсів класів

- Клас X розкриває (expose) інтерфейс (протокол з конструкторами, методами та, можливо, полями, які стають доступними для об’єктів, створених з інших класів для використання у створенні та комунікації з об’єктами X).
  - Java формалізує концепцію інтерфейсу, постачаючи зарезервоване слово interface, яке використовується для представлення типу без реалізації.
  - Java також постачає мовні інструменти для оголошення, реалізації та розширення інтерфейсів.
- Оголошення складається із заголовку інтерфейсу та тіла інтерфейсу.
  - За домовленістю назва інтерфейсів починається з великої літери, як і кожне слово в послідовності, що складає назву.
  - Багато назв інтерфейсів мають суфікс able в кінці (Callable, Comparable, Cloneable, Iterable, Runnable, Serializable)
-

```
interface Drawable
{
   int RED = 1;   // For simplicity, integer constants are used. These constants are
   int GREEN = 2; // not that descriptive, as you will see.
   int BLUE = 3;
   int BLACK = 4;
   void draw(int color);
}
```

Оголошення інтерфейсу, його полів та методів

- Ви можете дописувати public перед інтерфейсом, щоб зробити його доступним для коду ззовні пакету.
  - Інакше інтерфейс буде доступним лише всередині пакету, в якому він оголошений.
- Також можна приписувати спереду abstract для підкреслення абстрактності інтерфейсу.
  - Оскільки інтерфейс абстрактний за замовчуванням, це надлишковий код.
- Поля інтерфейсу неявно оголошуються public, static, final.
  - Оскільки ці поля константні, їх потрібно явно ініціалізувати; інакше компілятор повідомить про помилку
- Методи інтерфейсу неявно оголошуються public та abstract.
  - Оскільки ці методи мають бути методами екземпляру, не оголошуйте їх статичними, оскільки компілятор повідомить про помилку.
- Drawable ідентифікує тип, який задає, ***що*** робити (рисувати), але ***не говорить як*** це робити.
  - Деталі реалізації залишаються класам, які будуть реалізувати (implement) цей інтерфейс.
  - Екземпляри таких класів відомі як drawables, оскільки вони знають, як рисувати.

```
class Point implements Drawable
{
   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 + ")";
   }

   @Override
   public void draw(int color)
   {
      System.out.println("Point drawn at " + toString() + " in color " + color);
   }
}
```

Реалізація інтерфейсів

- Інтерфейс, який не оголошує членів, називають *marker interface* або *tagging interface*.
  - Він пов’язує метадані з класом.
  - Наприклад, наявність marker/tagging інтерфейсу Cloneable передбачає, що екземпляри класу, що його реалізує, можуть поверхнево клонуватись.
  - RTTI використовується для відстеження того, що клас об’єкту реалізує marker/tagging interface.
  - Наприклад, коли метод clone() класу Object detects, через RTTI, що виклик екземпляру класу реалізує Cloneable, він поверхнево клонує цей об’єкт.
- Сам по собі інтерфейс непотрібний.
  - Вигоду дає його реалізація за допомогою класу (зарезервоване слово implements)

Реалізація інтерфейсів

```
class Circle extends Point implements Drawable
{
   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
   public void draw(int color)
   {
      System.out.println("Circle drawn at " + super.toString() +
                         " with radius " + toString() + " in color " + color);
   }
}
```

- Зауважте, що кожен з класів Point та Circle реалізує цей інтерфейс, дописуючи «implements Drawable» до своїх заголовків.
  - Для реалізації інтерфейсу клас ***повинен*** задавати для кожного заголовку методу в інтерфейсі власний метод з також сигнатурою та типом повернення, а також кодом в його тілі.
  - Не забувайте, що методи інтерфейсу неявно оголошені публічними. Інакше компілятор повідомить про помилку через спробу присвоєння слабшого режиму доступу реалізованому методу.
- Коли клас реалізує інтерфейс, клас наслідує його константи та заголовки методів (переозначення за допомогою @Override).
  - Це називають ***наслідуванням інтерфейсу (interface inheritance)***.

- Виявляється, що заголовок Circle не потребує реалізації Drawable.
  - Якщо прибрати «implements Drawable», Circle успадкує метод draw() класу Point.
  - Але все-одно буде вважатись Drawable, переозначає він метод, чи ні.
- Інтерфейс задає тип, чиї значення є об’єктами, класи яких реалізують інтерфейс, а поведінка визначається інтерфейсом.
  - Тому можна присвоїти об’єкту посилання на змінну інтерфейсного типу. Наприклад:

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

  - Оскільки екземпляри Point та Circle є drawables, можна присвоювати посилання на об’єкти класів Point та Circle змінним (у т. ч. елементам масиву) типу Drawable. Вивід:

```
Point drawn at (10, 20) in color 1
Circle drawn at (10, 20) with radius 30 in color 1
```

Оголошення інтерфейсу Fillable

```
interface Fillable
{
   int RED = 1;
   int GREEN = 2;
   int BLUE = 3;
   int BLACK = 4;
   void fill(int color);
}
public static void main(String[] args)
{
   Drawable[] drawables = new Drawable[] { new Point(10, 20),
                                           new Circle(10, 20, 30) };
   for (int i = 0; i < drawables.length; i++)
      drawables[i].draw(Drawable.RED);
      Fillable[] fillables = new Fillable[drawables.length];
      for (int i = 0; i < drawables.length; i++)
      {
         fillables[i] = (Fillable) drawables[i];
         fillables[i].fill(Fillable.GREEN);
      }
   }
```

- Інтерфейс Drawable корисний для прорисовки контуру фігури (shape).
  - Припустимо, що потрібно виконати заливку.
- Можна оголосити, що класи Point та Circle реалізують обидва інтерфейси, задавши «implements Drawable, Fillable».
- Тоді можна змінити метод main() так, щоб звертатись до drawables, як і до *fillables.*

Множинні інтерфейси

```
interface A
{
   int X = 1;
   void foo();
}

interface B
{
   int X = 1;
   int foo();
}

class Collision implements A, B
{
   @Override
   public void foo();

   @Override
   public int foo() { return X; }
}
```

- Реалізація множинних інтерфейсів може призводити до name collisions – компілятор повідомить про помилки.
  - Припустимо, що інтерфейси A і B оголошують константу X.
  - Оскільки кожна константа має однаковий тип і значення, компілятор повідомить про помилку в другому методі foo() класу Collision, оскільки не знає, який X наслідується.
  - Крім того, другий foo() уже вважається оголошеним раніше!

```
Collision.java:19: error: method foo() is already defined in class Collision
   public int foo() { return X; }
              ^
Collision.java:13: error: Collision is not abstract and does not override abstract method foo() in B
class Collision implements A, B
^
Collision.java:16: error: foo() in Collision cannot implement foo() in B
   public void foo();
               ^
  return type void is not compatible with int
Collision.java:19: error: reference to X is ambiguous, both variable X in A and variable X in B
match
   public int foo() { return X; }
                             ^
4 errors
```

Розширення інтерфейсів

```
interface Colors
{
   int RED = 1;
   int GREEN = 2;
   int BLUE = 3;
   int BLACK = 4;
}

interface Drawable extends Colors
{
   void draw(int color);
}

interface Fillable extends Colors
{
   void fill(int color);
}
```

- Аналогічно до підкласів та суперкласів говорять про підінтерфейси (*subinterface) та суперінтерфейси*.
  - Називається *interface inheritance*.
  - Наприклад, дублювання констант color у Drawable та Fillable призводить до name collisions, якщо задавати лише назву змінної при реалізації в класі.
- Для уникнення цих name collisions, дописуйте спереду назву інтерфейсу та оператор «.» або розміщайте ці константи у власному інтерфейсі, щоб Drawable та Fillable розширювали його.

Для компілятора останній варіант – не проблема, оскільки існує єдина копія констант (у Colors), а можливості name collision немає.

Розширення декількох інтерфейсів

- Якщо клас може реалізувати декілька інтерфейсів (список інтерфейсів через кому після «implements»), розширювати декілька інтерфейсів заданий інтерфейс теж повинен уміти.

```
interface A
{
   int X = 1;
}

interface B
{
   double X = 2.0;
}

interface C extends A, B
{
}
```

Лістинг скомпілюється, незважаючи на те, що C наслідує дві одноіменні константи X різного типу та значень.
Проте при реалізації, намагаючись отримати доступ до X, як і раніше, виникне name collision.

```
class Collision implements C
{
   public void output()
   {
      System.out.println(X); // Which X is accessed?
   }
}
```

Навіщо використовувати інтерфейси?

- ***Інтерфейси в Java були створені з метою дати розробникам максимальну гнучкість при проектуванні додатків, відокремлюючи інтерфейс від реалізації.***
- При гнучкій (*agile) розробці ПЗ гнучке кодування дуже важливе*.
  - Зміна вимог на наступній ітерації може призвести до зміни реалізації, змушуючи переписувати значні частини коду, сповільнюючи розробку.
- Інтерфейси допомагають досягти гнучкості, відділивши інтерфейс від реалізації.
  - Наприклад, метод main() у класі Graphics створює масив об’єктів з класів, що є підкласами Shape, а потім проходить по цих об’єктах, викликаючи відповідний метод draw().
  - Єдиними об’єктами, що можна нарисувати, є об’єкти, суперкласом яких буде Shape.

Навіщо використовувати інтерфейси?

- Припустимо, існує ієрархія класів, що моделює електронні компоненти.
  - Кожен компонент має власне позначення, що дозволяє відображати його на схемі.
  - Можливо, Ви хочете додати можливість прорисовки для кожного класу, що рисує позначення компоненту.
- Можна розглянути Shape у якості суперкласу ієрархії класів електронних компонент.
  - Електронні компоненти не є фігурами (shape), тому немає смислу коренем робити Shape.
  - Але можна зробити, щоб клас кожного компоненту реалізовував інтерфейс Drawable, який дозволяє додавати вирази, які інстанціюють ці класи в масиві drawables у методі main().
  - Це легально, оскільки ці екземпляри є drawables.
  - ***По можливості потрібно завжди задавати інтерфейси замість класів у Вашому коді, щоб він залишався адаптивним до змін.***

Розглянемо простий приклад

- Складається з інтерфейсу java.util.List та його реалізацій java.util.ArrayList та java.util.LinkedList.

```
ArrayList<String> arrayList = new ArrayList<String>();
void dump(ArrayList<String> arrayList)
{
   // suitable code to dump out the arrayList
}
```

- Використовується generics-based параметризований тип (далі в лекції) для ідентифікації виду об’єкту, що зберігається в екземплярі ArrayList.
  - У даному випадку, зберігають об’єкти типу String.
- Приклад негнучкий, оскільки жорстко прив’язує клас ArrayList до кількох locations.
  - Це змушує розробника думати не про списки загалом, а про конкретно array lists.
  - Коли змінюються вимоги або *profiling вказує на проблеми з продуктивністю*, передбачається використання зв’язних списків (LinkedList).
  - Бажано внести мінімальну кількість змін у код для виконання нової вимоги.
- Навпаки, велика кодова база може потребувати набагато більше змін.
  - Хоча досить змінити ArrayList на LinkedList для виконання вимог компілятора, проте це може знадобитись зробити в багатьох місцях коду.

- Розробники обмежені часом при рефакторингу коду для адаптації до LinkedList.
  - Простіше було б переписати приклад з еквівалентними константами, тобто використати інтерфейси та задати ArrayList в одному місці.

```
List<String> list = new ArrayList<String>();
void dump(List<String> list)
{
   // suitable code to dump out the list
}
```

  - Код набагато гнучкіший.
  - Якщо зміна вимог чи результатів профілювання передбачає заміну ArrayList на LinkedList, просто замініть Array на Linked.
  - Не потрібно навіть змінювати parameter name.

Зауважте!

- Java постачає інтерфейси та абстрактні класи для опису *абстрактних типів* (які не можуть бути інстанційованими).
  - Абстрактні типи представляють абстрактні поняття (наприклад, drawable та shape), а екземпляри таких типів не будуть мати смислу.
  - Інтерфейси надають гнучкість через нестачу реалізації.
  - Інтерфейси не прив’язані до конкретної ієрархії класів, проте можуть бути реалізовані в конкретній ієрархії.
    - На противагу цьому, абстрактні класи підтримують реалізацію, але можуть бути genuinely abstract.
    - Проте вони обмежені верхніми рівнями ієрархії класів.
- Інтерфейси та абстрактні класи можуть використовуватись разом.
  - Наприклад, пакет java.util постачає інтерфейси List, Map та Set, абстрактні класи AbstractList, AbstractMap та AbstractSet, які забезпечують реалізацію каркасу цих інтерфейсів**.**
  - Реалізації каркасів (skeletal implementations) спрощують створення власних реалізацій інтерфейсів відповідно до конкретних вимог.
  - Якщо вони не відповідають вимогам, можна, щоб опційний клас напряму реалізовував інтерфейс.

У перших версіях Java всі методи в інтерфейсі були абстрактними

- Тепер в інтерфейсі допускаються 2 різновиди методів з конкретною реалізацією: статичні та за замовчуванням.
- **Уявлення про статичні методи в інтерфейсах змінилось.**
  - **Зокрема, в інтерфейсах корисно визначати фабричні методи.**
  - **В Інтерфейсі** IntSequence **може бути оголошено статичний метод** digitsOf(), який формує послідовність цифр заданого зразка:
  - IntSequence digits = IntSequence.digitsOf(1729);
  - **Метод повертає екземпляр деякого класу, що реалізує інтерфейс** IntSequence**, але у викликаючому коді назва класу не важлива.**

```
public interface IntSequence {
  ...
  public static IntSequence digitsOf(int n) {
    return new DigitSequence(n);
  }
}
```

Методи за умовчанням

- Можна надати для будь-якого інтерфейсного методу.
  - **Такий метод потрідбно позначити модифікатором доступу** default.
  - **Метод** hasNext() **може бути переозначений або успадкуватись.**

```
public interface IntSequence {
  default boolean hasNext() { return true; }
    // По умолчанию последовательности бесконечны
  int next();
}
```

- **Приклад: інтерфейс** Collection та клас, що його реалізує:
  - public class Bag implements Collection
  - **Починаючи з Java 8 в інтерфейс додали метод** stream(), що не є методом за замовчуванням
  - **Клас** Bag більше не скомпілюється, оскільки не реалізує новий метод з інтерфейсу Collection.
  - **Впровадження в інтерфейс методу НЕ за умовчанням порушує сумісніть на рівні первинного коду.**

Припустимо, що цей клас не перекомпілюється

- Використовується старий архівний JAR-файл, який його містить.
  - Клас все ще завантажується, екземпляри класу Bag все ще можна конструювати, оскільки впровадження методу в інтерфейс сумісне на рівні двійкового коду.
  - Проте при виклику методу stream () для екземпляру класу Bag виникає AbstractMethodError.
- Проблеми усуне оголошення методу stream () як default.
  - Якщо клас зававнтажується без перекомпіляції, а метод stream() викликається для екземпляру класу Bag, то такий виклик відбувається за посиланням Collection.stream().

Вирішення конфліктів з методами за замовчуванням

- **Нечаста проблема:** клас реалізує кілька інтерфейсів, в деяких з них наявні методи (за умовчанням) з однаковими сигнатурами.
  - **Нехай існує інтерфейс** Person **з методом за умовчанням** getId():

```
public interface Person {
  String getName();
  default int getId() { return 0; }
}
```

  - **Також є інтерфейс** Identified з таким же методом за замовчуванням**:**

```
public interface Identified {
    default int getId() { return Math.abs(hashCode()); }
}
```

Що буде, якщо клас буде реалізовувати обидва інтерфейси?

```
public class Employee implements Person, Identified {
  ...
}
```

- Наслідуються обидва методи getId(), проте критерію вибору компілятор не має.
  - З'явиться повідомлення про помилку
  - Потрібно реалізувати метод getId(), який прибере неоднозначність:

```
public class Employee implements Person, Identified {
  public int getId() { return Identified.super.getId(); }
  ...
}
```

Нехай Identified не надає власну реалізацію getId() за замовчуванням

```
interface Identified {
   int getId();
}
```

- Чи може клас Employee успадкувати метод за умовчанням з інтерфейсу Person?
  - Якщо хоча б один з інтерфейсів надає реалізацію методу, компілятор повідомить про помилку.
  - Неоднозначність повинен вирішити програміст.
  - Якщо реалізацій за умовчанням немає, конфлікту не виникне, - потрібно реалізувати метод або оголосити клас абстрактним.

```
interface Directions
{
   int NORTH = 0;
   int SOUTH = 1;
   int EAST = 2;
   int WEST = 3;
}
```

```
public class TrafficFlow implements Directions
{
   public static void main(String[] args)
   {
      showDirection((int) (Math.random()* 4));
   }

   static void showDirection(int dir)
   {
      switch (dir)
      {
         case NORTH: System.out.println("Moving north"); break;
         case SOUTH: System.out.println("Moving south"); break;
         case EAST : System.out.println("Moving east"); break;
         case WEST : System.out.println("Moving west");
      }
   }
}
```

- Інтерфейс має використовуватись лише для оголошення типу
  - Проте деякі розробники порушують цей принцип, використовуючи інтерфейси лише для експорту констант.
  - Такі інтерфейси називаються константними.
  - Мета їх використання – уникнення префіксів перед назвою константи, наприклад, Math.PI, де PI – константа з класу java.lang.Math.

Клас TrafficFlow реалізує інтерфейс Directions з єдиною метою: не задавати Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST.

Статичний імпорт

- Константи є просто деталями реалізації, які не мають показуватись в інтерфейсі.
  - Також може бути майбутня проблема: навіть якщо клас вже не використовує константи, вони мають залишитись в інтерфейсі для забезпечення binary compatibility.
  - Java 5 представила альтернативу - статичний імпорт, який дозволяє імпортувати статичні члени класу.
- Реалізуються невеликою зміну синтаксису:
  import static *packagespec* . *classname* . (*staticmembername* | *);
  - Оператор статичного імпорту дописує static після import, а далі специфікує список пакетів та підпакетів, розділений крапками.
- **Обережно!** Розміщення будь-чого відмінного від package statement, import/static import statements та коментарів над виразом для статичного імпорту змушує компілятор повідомляти про помилку.
- Задається окрема назва статичного члена, щоб імпортувати лише це ім’я.
  - import static java.lang.Math.PI; // Import the PI static field only.
  - import static java.lang.Math.cos; // Import the cos() static method only.

Можливі проблеми зі статичним імпортом

- Для імпорту всіх імен статичних членів задається підстановка (wildcard).
  - import static java.lang.Math.*; // Import all static members from Math.
  - Тепер можна звертатись до статичних членів без назви класу.
  - System.out.println(cos(PI));
- Використання кількох виразів для статичного імпорту може призвести до конфлікту іменування, що веде до помилок компілятора.
  - Наприклад, пакет geom містить клас Circle зі статичний полем PI. Тепер задамо згори файлу

```
import static java.lang.Math.*;
import static geom.Circle.*;
```

  - Та викличемо System.out.println(PI); десь у коді.
  - Компілятор повідомить про помилку, оскільки не знає, відноситься PI до Math чи до Circle.
- Зачасте використання статичних імпортів може зробити код нечитабельним та unmaintainable.
  - Оглядачі коду довго шукатимуть, з якого класу взято статичний член, особливо при повному імпорті за допомогою *.
- Також статичні імпорти забруднюють простір імен у коді великою кількістю імпортованих статичних членів.
  - З часом можна отримати конфлікти імен, які важко вирішити.

ДЯКУЮ ЗА УВАГУ!

Наступне запитання: узагальнені типи Java