УЗАГАЛЬНЕНІ ТИПИ В МОВІ JAVA

Питання 3.3.

Вступ до дженериків

- Дженерики представлені у Java 5 для оголошення та використання type-agnostic класів та інтерфейсів.
  - Працюючи з Collections Framework, вони допоможуть уникати java.lang.ClassCastExceptions.
- Стандартна бібліотека класів також містить узагальнені класи, які не мають нічого спільного з цим фреймворком:
  - java.lang.Class,
  - java.lang.ThreadLocal,
  - java.lang.ref.WeakReference.
- Collections Framework дозволяє зберігати об’єкти в різного роду контейнерах (т. з. колекціях), а потім відбирати ці об’єкти.
  - Зокрема, можна їх зберігати в списку (list), сеті (множині) або карті (map).
  - Можна робити відбір одного об’єкта або ітерувати по колекції та відбирати (retrieve) всі об’єкти.
- До Java 5 не було можливості запобігти тому, щоб колекція містила об’єкти змішаних типів.
  - Компілятор не перевіряв тип об’єкту на його відповідність перед додаванням у колекцію.
  - Нестача перевірки статичних типів призводила до ClassCastExceptions.

Нестача безпеки типів, що веде до ClassCastException при виконанні

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

class Employee
{
   private String name;

   Employee(String name)
   {
      this.name = name;
   }
   String getName()
   {
      return name;
   }
}
```

```
public class TypeSafety
{
   public static void main(String[] args)
   {
      List employees = new ArrayList();
      employees.add(new Employee("John Doe"));
      employees.add(new Employee("Jane Smith"));
      employees.add("Jack Frost");
      Iterator iter = employees.iterator();
      while (iter.hasNext())
      {
         Employee emp = (Employee) iter.next();
         System.out.println(emp.getName());
      }
   }
}
```

- Спроба зведення "Jack Frost" до типу Employee закінчиться ClassCastException.
  - ClassCastException виникає через припущення однорідності списку, а насправді список гетерогенний, оскільки може зберігати будь-який Object.

Нестача безпеки типів призводить до помилки компіляції

```
public class TypeSafety
{
   public static void main(String[] args)
   {
      List<Employee> employees = new ArrayList<Employee>();
      employees.add(new Employee("John Doe"));
      employees.add(new Employee("Jane Smith"));
      employees.add("Jack Frost");
      Iterator<Employee> iter = employees.iterator();
      while (iter.hasNext())
      {
         Employee emp = iter.next();
         System.out.println(emp.getName());
      }
   }
}
```

- Модифікуємо метод main().
  - Продемонстровано основну рису дженериків – параметризацію типів (у кутових дужках описано список типів, легальних для даного контексту).
  - Наприклад, java.util.List<Employee> вказує на те, що лише об'єкти типу Employee можуть зберігатись у List.
- Якщо спробуєте скомпілювати лістинг, компілятор повідомить про помилку при виклику employees.add("Jack Frost");.
  - Це повідомлення говорить, що компілятор не може знайти метод add(java.lang.String) в інтерфейсі java.util.List<Employee>.
  - На відміну до pre-generics інтерфейсу List, який оголошував метод add(Object), узагальнений параметр методу add() інтерфейсу List відображає параметричну назву типу інтерфейсу.
  - Наприклад, List<Employee> передбачає метод add(Employee)

Також Iterator<Employee> вказує на те, що iterator() повертає ітератор, чий метод next() повертає лише об'єкти типу Employee. Немає потреби зводити значення, яке повертається від iter.next(), до типу Employee, оскільки компілятор це робить сам.

Узагальнені типи (Generic Types)

- Узагальненим типом є клас або інтерфейс, які вводять сімейство параметризованих типів, оголошуючи formal type parameter list.

```
class identifier<formal_type_parameter_list> {}
interface identifier<formal_type_parameter_list> {}
```

  - Наприклад, List<E> - узагальнений тип, де List – інтерфейс, а type parameter E вказує на тип елементів списку.
  - Аналогічно, Map<K, V> - узагальнений тип, де Map – інтерфейс, а type parameters K і V визначають типи ключа та значення карти.
  - При оголошенні узагальненого типу прийнято задавати одну велику букву в якості назви параметру.
  - Крім того, імена мають бути значущими: E – елемент, T – тип, K – ключ, V – значення.

- Параметризовані типи є екземплярами узагальнених типів.
  - Кожен параметризований тип заміняє параметри узагальненого типу (generic type's type parameters) на назви типів.
  - Наприклад, List<Employee> (List of Employee) та List<String> (List of String) є прикладами параметризованих типів на основі List<E>.
  - Аналогічно, Map<String, Employee> є прикладом параметризованого типу на основі Map<K, V>.

Аргумент фактичного типу (actual type argument)

- Назва типу, яка заміняє тип-параметр (type parameter), називається actual type argument (фактичним типом аргументу).
- Дженерики підтримують 5 видів actual type arguments:
  - ***Конкретний тип (Concrete type)***: Назва класу або інтерфейсу передається параметричному типу. Наприклад, List<Employee> list;.
  - ***Конкретний параметризований тип (Concrete parameterized type):*** List<List<String>> nameLists;
  - ***Тип-масив (Array type):*** масив передається в параметризований тип. Наприклад, List<String[]> countries;
  - ***параметризований тип (Type parameter):*** class X<E> { List<E> queue; },
    - параметризований тип E параметризованого типу X передається в параметризований тип E списку List.
  - ***Маска (Wildcard)***: Знак «?» передається в параметризований тип.
    - Наприклад, List<?> list;

Сирі та узагальнені типи

- Узагальнений (generic) тип також визначає «сирий» (raw) тип, який є узагальненим типом без його типів-параметрів.
  - Наприклад, «сирим» типом List<Employee> є List.
  - «Сирі» типи не є узагальненими і можуть містити будь-який Object.
- Зауважте! Java дозволяє перемішувати «сирі» та узагальнені типи з метою підтримки коду, написаного до появи дженериків.
  - Проте компілятор виводить warning message, коли знаходить в коді «сирий» тип.

Оголошення та використання власних узагальнених типів

```
public class Queue<E>
{
   private E[] elements;
   private int head, tail;

   @SuppressWarnings("unchecked")
   Queue(int size)
   {
      if (size < 2)
         throw new IllegalArgumentException("" + size);
      elements = (E[]) new Object[size];
      head = 0;
      tail = 0;
   }

   void insert(E element) throws QueueFullException
   {
      if (isFull())
         throw new QueueFullException();
      elements[tail] = element;
      tail = (tail + 1) % elements.length;
   }
```

- Оголосити власний узагальнений тип неважко.
  - Крім formal type parameter list, тип-дженерик з'являється в коді реалізації.
  - Queue реалізує чергу (first-in, first-out).
    - Елемент додається у хвіст і видаляється з голови.
  - Черга порожня, коли голова дорівнює хвосту, і повна, коли при довжині черги n хвіст вкзує на n – 1 елемент.
- Зауважте, що параметричний тип E з Queue<E> з'являється у вихідному коді.
  - Наприклад, при оголошенні масиву для позначення типу елементів.
  - Також тип E є типом параметру методу insert() та типом повернення методу remove().
  - Ще тип E з'являється у виразі elements = (E[]) new Object[size];

Продовження класу Queue

```
E remove() throws QueueEmptyException
{
   if (isEmpty())
      throw new QueueEmptyException();
   E element = elements[head];
   head = (head + 1) % elements.length;
   return element;
}

boolean isEmpty()
{
   return head == tail;
}

boolean isFull()
{
   return (tail + 1) % elements.length == head;
}
```

- Зведення до типу E[] призводить до попередження компілятора про те, що це зведення being unchecked.
- Компілятор стурбований понижуючим зведенням від Object[] до E[] та можливим порушенням безпеки типів, оскільки в Object[] може зберігатись будь-який тип.
  - У прикладі обробки попередження немає, оскільки об’єкт з відмінним від E типом не може з’явитись у масиві E[].
  - Оскільки попередження несуттєве в даному контексті, воно пригнічується анотацією @SuppressWarnings("unchecked") перед конструктором.

```
    public static void main(String[] args)
       throws QueueFullException, QueueEmptyException
    {
       Queue<String> queue = new Queue<String>(6);
       System.out.println("Empty: " + queue.isEmpty());
       System.out.println("Full: " + queue.isFull());
       System.out.println("Adding A");
       queue.insert("A");
       System.out.println("Adding B");
       queue.insert("B");
       System.out.println("Adding C");
       queue.insert("C");
       System.out.println("Adding D");
       queue.insert("D");
       System.out.println("Adding E");
       queue.insert("E");
       System.out.println("Empty: " + queue.isEmpty());
       System.out.println("Full: " + queue.isFull());
       System.out.println("Removing " + queue.remove());
       System.out.println("Empty: " + queue.isEmpty());
       System.out.println("Full: " + queue.isFull());
       System.out.println("Adding F");
       queue.insert("F");
       while (!queue.isEmpty())
          System.out.println("Removing " + queue.remove());
       System.out.println("Empty: " + queue.isEmpty());
       System.out.println("Full: " + queue.isFull());
    }
}
```

Метод main() класу

- Будьте обережні з пригніченням unchecked warning.
  - Спочатку переконайтесь, що ClassCastException не може виникнути.

```
class QueueEmptyException extends Exception
{
}

class QueueFullException extends Exception
{
}
```

```
Empty: true
Full: false
Adding A
Adding B
Adding C
Adding D
Adding E
Empty: false
Full: true
Removing A
Empty: false
Full: false
```

```
Adding F
Removing B
Removing C
Removing D
Removing E
Removing F
Empty: true
Full: false
```

Межі параметричного типу (Type Parameter Bounds)

- Параметризований тип E в List<E>, а також параметризовані типи з Map<K, V>'s є прикладами необмежених (unbounded) типів-параметрів (type parameters).
  - Можна передавати будь-який аргумент фактичного типу to an unbounded type parameter.
- Інколи не потрібно обмежувати the kinds of actual type arguments, які можуть передаватись у параметризований тип.
  - Наприклад, потрібно оголосити клас, чиї екземпляри можуть містити лише екземпляри класів, що є підкласами абстрактного класу Shape (як Circle та Rectangle).
- Для обмеження actual type arguments можна задати верхню межу (upper bound) – тип, який слугує верхньою границею для типів, що можуть бути обраними в якості actual type arguments.

Верхня межа

- Верхня межа задається за допомогою зарезервованого слова extends перед назвою типу.
  - Наприклад, ShapesList<E extends Shape> вказує, що Shape – верхня межа.
  - Можна записати ShapesList<Circle>, ShapesList<Rectangle> і навіть ShapesList<Shape>, проте не ShapesList<String> (String – не підклас Shape).
- Можна присвоїти більш ніж одну верхню межу параметричному типу.
  - Тоді перша межа – клас або інтерфейс, а кожна додаткова верхня межа є інтерфейсом.
  - Використовується символ &.

```
abstract class Shape
{
}

class Circle extends Shape implements Comparable<Circle>
{
   private double x, y, radius;

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

   @Override
   public int compareTo(Circle circle)
   {
      if (radius < circle.radius)
         return -1;
      else
      if (radius > circle.radius)
         return 1;
      else
         return 0;
   }

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

Присвоєння кількох верхніх меж параметричному типу

- Клас Circle розширяє клас Shape та реалізує інтерфейс java.lang.Comparable, який використовується для встановлення природного порядку об’єктів типу Circle.
- Метод compareTo() інтерфейсу реалізує цю впорядкованість, повертаючи значення, що відображає порядок.
  - Від’ємне значення – поточний об’єкт має передувати об’єкту, переданому в метод compareTo() in some fashion.
  - Нуль – поточний об’єкт та аргумент методу однакові.
  - Додатне значення – поточний об’єкт має слідувати після аргументу методу.
- Переозначений метод compareTo() класу Circle порівнює два об’єкти типу Circle на основі їх радіусів.
  - Екземпляри класу Circle впорядковуються від меншого радіуса до більшого.

```
class SortedShapesList<S extends Shape & Comparable<S>>
{
   @SuppressWarnings("unchecked")
   private S[] shapes = (S[]) new Shape[2];
   private int index = 0;

   void add(S shape)
   {
      shapes[index++] = shape;
      if (index < 2)
         return;
      System.out.println("Before sort: " + this);
      sort();
      System.out.println("After sort: " + this);
   }

   private void sort()
   {
      if (index == 1)
         return;
      if (shapes[0].compareTo(shapes[1]) > 0)
      {
         S shape = (S) shapes[0];
         shapes[0] = shapes[1];
         shapes[1] = shape;
      }
   }

   @Override
   public String toString()
   {
      return shapes[0].toString() + " " + shapes[1].toString();
   }
}
```

- Клас SortedShapesList задає в якості списку параметрів <S extends Shape & Comparable<S>>.
  - The actual type argument, переданий в параметр S, має субкласувати Shape, а також реалізовувати інтерфейс Comparable.
  - Circle задовольняє обидва критерії.
  - В результаті компілятор не повідомляє про помилку, коли зустрічає інструкцію в методі main()
  - SortedShapesList<Circle> ssl = new SortedShapesList<Circle>();
- Верхня межа пропонує додаткову статичну перевірку типу, яка гарантує, що параметричний тип дотримується границь.
  - Підтверджує безпечні виклики методів.
  - Наприклад, метод sort() може викликати метод compareTo() інтерфейсу Comparable.

Виведення програми

```
public class SortedShapesListDemo
{
   public static void main(String[] args)
   {
      SortedShapesList<Circle> ssl = new SortedShapesList<Circle>();
      ssl.add(new Circle(100, 200, 300));
      ssl.add(new Circle(10, 20, 30));
   }
}
```

- Вивід додатку, який показує два об’єкта типу Circle, що зберігаються в порядку зростання радіусу:

```
Before sort: (100.0, 200.0, 300.0) (10.0, 20.0, 30.0)
After sort: (10.0, 20.0, 30.0) (100.0, 200.0, 300.0)
```

Область видимості параметричних типів (Type Parameter Scope)

- Областю видимості параметризованого типу є його узагальнений тип, except where masked (hidden).
  - Область видимості включає formal type parameter list of which the type parameter is a member.
  - Наприклад, область видимості S у SortedShapesList<S extends Shape & Comparable<S>> є all of SortedShapesList and the formal type parameter list.
- Можливо замаскувати параметричний тип, оголосивши параметричний тип з такою ж назвою у nested type's formal type parameter list.
  - Наприклад, лістинг маскує зовнішній (enclosing) параметризований тип T класу
  - Referencing T зсередини EnclosedClass відноситься до обмеженого T, а не необмеженого T, що передається в EnclosingClass.

```
class EnclosingClass<T>
{
   static class EnclosedClass<T extends Comparable<T>>
   {
   }
}
```

Потреба в масках (підстановочних типах, Wildcards)

```
import java.util.ArrayList;
import java.util.List;

public class OutputList
{
   public static void main(String[] args)
   {
      List<String> ls = new ArrayList<String>();
      ls.add("first");
      ls.add("second");
      ls.add("third");
      outputList(ls);
   }

   static void outputList(List<Object> list)
   {
      for (int i = 0; i < list.size(); i++)
         System.out.println(list.get(i));
   }
}
```

- Припустимо, створили список рядків (List of String) і хочемо вивести його.
  - Оскільки можна створити List of Employee та інші види списків, Ви забажаєте, щоб метод виводив список довільних об'єктів (List of Object).

```
OutputList.java:12: error: method outputList in class OutputList cannot be applied to given types;
      outputList(ls);
      ^
  required: List<Object>
  found: List<String>
  reason: actual argument List<String> cannot be converted to List<Object> by method invocation
conversion
1 error
```

Отримана помилка слідує з незнання базового правила узагальнених типів: для заданого підтипу ***x типу y, та даного «сирого» типу G, G<x> НЕ є підтипом G<y>***.

specialized kind

- Підтип, в основному, є specialized kind свого супертипу.
  - Наприклад, Circle є видом Shape, а String є видом Object.
- Така поліморфна поведінка також застосовується до пов’язаних параметризованих типів з однаковими параметрами типу.
  - Наприклад, List<Object> is a specialized kind of java.util.Collection<Object>.
- Проте така поліморфна поведінка не застосовується до множинних параметризованих типів (multiple parameterized types), які відрізняються лише одним типом параметру, що є підтипом іншого типу параметру.
  - Наприклад, List<String> is ***not*** a specialized kind of List<Object>.

Неполіморфність параметризованих типів з різними параметрами

- Приклад того, чому параметризовані типи з різними типами параметрів не поліморфні:

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
Lo.add(new Employee());
String s = ls.get(0);
```

- Не скопмпілюється, оскільки порушує безпеку типів.
  - Буде викинуто виключення ClassCastException через неявне зведення типу (implicit cast) до String в останньому рядку.
  - Спершу інстанціюємо List of String, а потім у другому рядку зведемо посилання до базового типу (upcast) – List of Object.
  - У третьому рядку додамо новий об’єкт типу Employee в List of Object.
  - Далі отримуємо цей об’єкт за допомогою get() та намагаємось присвоїти його рядковій змінній. Проте Employee – не String.

Зауважте Хоч Ви не можете звести **(**upcast) List<String> до List<Object>, можна звести List<String> до сирого типу List, щоб interoperate with legacy code.

Узагальнені методи

- Припустимо, потрібен метод, що копіює список об’єктів будь-якого типу в інший List.
  - Метод void copyList(List<Object> src, List<Object> dest) матиме обмежену корисність, оскільки зможе копіювати лише списки, елементи яких типу Object.
  - Наприклад, неможливо скопіювати List<Employee>.
- Якщо хочете передати початковий та кінцевий списки, чиї елементи будуть довільного типу (проте узгодженого), необхідно задати підстановочний символ (wildcard character).
  - Наприклад, візьмемо метод класу copyList(), який приймає колекції з довільним типом об’єктів у якості аргументів

```
static void copyList(List<?> src, List<?> dest)
{
   for (int i = 0; i < src.size(); i++)
      dest.add(src.get(i));
}
```

Проте є помилка

- Список параметрів методу коректний, проте компілятор виведе помилку при досягненні виразу dest.add(src.get(i));:

```
CopyList.java:19: error: no suitable method found for add(Object)
         dest.add(src.get(i));
             ^
    method List.add(int,CAP#1) is not applicable
      (actual and formal argument lists differ in length)
    method List.add(CAP#1) is not applicable
      (actual argument Object cannot be converted to CAP#1 by method invocation conversion)
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 error
```

- Передбачається, що метод copyList() – частина класу CopyList.
- Виклик методу dest.add(src.get(i)) порушує безпеку типів.

Проте є помилка

- Оскільки ? передбачає, що об’єкт будь-якого типу може бути елементом списку, можливо, що типи елементів початкового та кінцевого списків не співпадають (неузгоджені).
  - Наприклад, ми хочемо скопіювати List of String в List of Employee, що порушує безпеку типів.
  - Якби таке копіювання було дозволено, викидались би виключення типу ClassCastException при спробі отримання кінцевого списку.
- Проблему можна вирішити так:

```
static void copyList(List<? extends String> src,
                     List<? super String> dest)
{
   for (int i = 0; i < src.size(); i++)
      dest.add(src.get(i));
}
```

Цей метод демонструє wildcard аргумент, в якому Ви можете підтримувати верхню межу або (на відміну від type parameter) понижувати границю для обмеження типів, що можуть передаватись в якості actual type arguments to the generic type.

- Інтерпретуємо «? extends String»: будь-який actual type argument, що буде String або підкласом цього типу, можна передавати.
- Інтерпретуємо «? super String»: будь-який actual type argument, що буде String або його суперкласом, можна передавати.
  - Оскільки String не можна субкласувати, вихідний список може складатись лише зі String-об'єктів, а кінцевий – з об'єктів типу String або Object.
  - Проблема копіювання списків з елементами довільного типу вирішується за допомогою узагальнених методів (*generic method) – методів класу або екземпляру з* type-generalized реалізацією).
- Синтаксичне вираження узагальнених методів:
- *<formal_type_parameter_list> return_type identifier*(*parameter_list*)
  - *formal_type_parameter_list* такий же, як і при задаванні узагальненого типу: він містить type parameters з опційними межами (optional bounds).
  - Типи параметру можуть з'являтись у вигляді *return_type* методу, а також у *parameter_list*.
  - Компілятор визначає фактичні аргументи з контексту, в якому викликається метод.

- У Collections Framework часто використовуються узагальнені методи.
  - Наприклад, клас java.util.Collections постачає метод
  - public static <T extends Object & Comparable<? superT>> T min(Collection<? extends T> coll) для повернення мінімального елемента в даній колекції згідно з природним порядком його елементів.
- Можна просто сконвертувати copyList() в узагальнений метод, додаючи спереду return type <T> та замінивши кожний підстановочний символ на T.
- У результаті заголовок методу
  - <T> void copyList(List<T> src, List<T> dest),
  - а наступний лістинг представляє його вихідний код як частину додатку, яка копіює List of Circle в інший List of Circle.

Оголошення та використання узагальненого методу copyList()

```
import java.util.ArrayList;
import java.util.List;

class Circle
{
   private double x, y, radius;

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

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

```
public class CopyList
{
   public static void main(String[] args)
   {
      List<String> ls = new ArrayList<String>();
      ls.add("A");
      ls.add("B");
      ls.add("C");
      outputList(ls);
      List<String> lsCopy = new ArrayList<String>();
      copyList(ls, lsCopy);
      outputList(lsCopy);
      List<Circle> lc = new ArrayList<Circle>();
      lc.add(new Circle(10.0, 20.0, 30.0));
      lc.add(new Circle (5.0, 4.0, 16.0));
      outputList(lc);
      List<Circle> lcCopy = new ArrayList<Circle>();
      copyList(lc, lcCopy);
      outputList(lcCopy);
   }

   static <T> void copyList(List<T> src, List<T> dest)
   {
      for (int i = 0; i < src.size(); i++)
         dest.add(src.get(i));
   }

   static void outputList(List<?> list)
   {
      for (int i = 0; i < list.size(); i++)
         System.out.println(list.get(i));
      System.out.println();
   }
}
```

Компілятор використовує type inference algorithm to infer типи аргументів узагальнених методів з контексту, в якому метод було викликано

Результати виводу

```
A
B
C

A
B
C

(10.0, 20.0, 30.0)
(5.0, 4.0, 16.0)

(10.0, 20.0, 30.0)
(5.0, 4.0, 16.0)
```

- Компілятор використовує type inference algorithm to infer типи аргументів узагальнених методів з контексту, в якому метод було викликано
  - Наприклад, компілятор визначає, що copyList(ls, lsCopy); копіює List of String в інший List of String.
  - Аналогічно він визначає, що copyList(lc, lcCopy); копіює List of Circle в інший List of Circle.
- Без цього алгоритму потрібно було б задавати ці аргументи так:
  - CopyList.<String>copyList(ls, lsCopy);
  - CopyList.<Circle>copyList(lc, lcCopy);

Узагальнені конструктори

- Generic та non-generic класи можуть оголошувати узагальнені конструктори, в яких конструктор має список параметрів формального типу (formal type parameter list).
  - Наприклад, можна оголосити наступний неузагальнений клас з узагальненим конструктором:

```
public class GenericConstructorDemo
{
   <T> GenericConstructorDemo(T type)
   {
      System.out.println(type);
   }
   public static void main(String[] args)
   {
      GenericConstructorDemo gcd = new GenericConstructorDemo("ABC");
      gcd = new GenericConstructorDemo(new Integer(100));
   }
}
```

Виводить:
ABC
100.

Масиви та дженерики

- Після представлення узагальненого типу Queue<E> (слайд), було анонсовано пояснення того, чому записувалось
  - elements = (E[]) new Object[size]; замість більш компактного запису elements = new E[size];
- Через реалізацію дженериків у Java, неможливо задавати array-creation вирази, які включають
  - типи параметрів (type parameters, наприклад, new E[size] або new List<E>[50])
  - або actual type arguments (такі як new Queue<String>[15]).
- Якщо все ж так зробити, компілятор повідомить про помилку.
  - потрібно зрозуміти поняття реіфікації та коваріантності відносно масивів, а також поняття стирання (erasure), яке є центральним принципом в реалізації дженериків.

Реіфікація (*Reification*)

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

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

```
public class ReificationDemo
{
   public static void main(String[] args)
   {
      ColoredPoint[] cptArray = new ColoredPoint[1];
      Point[] ptArray = cptArray;
      ptArray[0] = new Point();
   }
}
```

- Представлення абстрактного так, наче воно конкретне.
  - Наприклад, відкриття доступу до адрес пам'яті для прямих маніпуляцій іншими мовними конструкціями.
- Масиви Java реіфіковані – вони знають типи своїх елементів (тип елементу зберігається internally) та можуть enforce їх під час виконання.
  - Спроба зберігати invalid елемент у масиві змушує віртуальну машину викидати виключення з класу java.lang.ArrayStoreException.

ColoredPoint[] cptArray = new Point[1]; не буде компілюватись, виникне ClassCastException під час виконання – масив знає, що присвоєння нелегальне.

Коваріантність

- Другий рядок (Point[] ptArray = cptArray;) легальний через коваріантність (масив посилань на супертип є супертипом масиву посилань на підтип).
  - У даному випадку масив посилань на Point є супертипом масиву посилань на ColoredPoint.

- Надмірне використання коваріантності небезпечне.
  - У третій лінії (ptArray[0] = new Point();) під час виконання виникає ArrayStoreException, осільки екземпляр Point не є екземпляром ColoredPoint.
  - Без цього виключення спроба доступу до неіснуючого члену color призводить віртуальну машину до краху.

На відміну від масивів, узагальнені типи параметрів не реіфіковані

- Вони не доступні під час виконання, оскільки викидаються після компіляції коду.
  - «Викидання типів параметрів» (“throwing away of type parameters”) – результат стирання (*erasure*), яке також задіює вставку зведення до підходящого типу, when the code isn’t type correct, and replacing type parameters by their upper bounds (such as Object).
- Компілятор виконує стирання, щоб дати можливість узагальненому коду взаємодіяти з legacy (nongeneric) кодом.
  - Він трансформує узагальнений код у звичайний (nongeneric runtime code).
  - Один з наслідків стирання – неможливо використовувати оператор instanceof з параметризованими типами окремо від необмежених (unbounded) підстановочних типів.
  - Наприклад, нелегально задавати
    - List<Employee> le = null;
    - if (le instanceof ArrayList<Employee>) {}.
  - Замість цього потрібно записувати
    - le instanceof ArrayList<?> (необмежена маска) або
    - le instanceof ArrayList (сирий тип, який використовується).

Задамо вираз, що створює масив, включаючи параметр типу або actual type argument.

- Чому це погано?
  - Розглянемо приклад, що має згенерувати ArrayStoreException замість ClassCastException, але цього не робить:

```
List<Employee>[] empListArray = new ArrayList<Employee>[1];
List<String> strList = new ArrayList<String>();
strList.add("string");
Object[] objArray = empListArray;
objArray[0] = strList;
Employee e = empListArray[0].get(0);
```

  - Припустимо, що перший рядок легальний.
  - Четвертий рядок присвоює empListArray масиву objArray.
  - Це коректно, оскільки масиви коваріантні, а стирання конвертує List<Employee>[] в List runtime type, і List subtypes Object.

Проблеми стирання та зведення типів у процесі компіляції

- Через стирання віртуальна машина не викидає ArrayStoreException при зустрічі інструкції objArray[0] = strList;.
  - Ви присвоюєте посилання на List масиву List[] під час виконання.
  - Проте це виключення буде викидатись, якби узагальнені типи були реіфіковані, because you'd then be assigning a List<String> reference to a List<Employee>[] array.
- Є проблема. Екземпляр List<String> зберігався в масиві, який може містити лише екземпляри List<Employee>.
  - Коли compiler-inserted оператор зведення спробує звести значення, що повертає empListArray[0].get(0), а саме "string", до типу Employee, то викинеться виключення ClassCastException.

ДЯКУЮ ЗА УВАГУ!

Наступне запитання: узагальнені типи Java