ОСНОВИ РОБОТИ З ВИНЯТКАМИ ТА ТВЕРДЖЕННЯМИ

Питання 1.5.

Що таке виняток?

- ***Виняток*** – це відхилення від нормальної поведінки ПЗ.
  - Наприклад, спроба відкрити неіснуючий файл для читання викликає виключення, якому неможливо запобігти.
- Можливий ***обхід винятків***.
  - Наприклад, додаток може виявляти відсутність файлу та змінювати хід роботи – повідомляти користувача про проблему.
- Винятки можуть виникати через погано написаний код.
  - Нехай додаток містить код, який отримує доступ до кожного елементу в масиві.
  - Через неуважність розробника може виникати спроба доступу в неіснуючий елемент, що призводить до появи виключення.
  - Такий тип винятків ***усувається*** шляхом написання коректного коду.
- Також можуть виникати ***неусувні виключення***, обійти які неможливо.
  - Наприклад, віртуальній машині не вистачає пам'яті, або вона не може знайти class-файл.
  - Такий тип виключень називають ***помилками*** (*error*).
  - Це настільки серйозне виключення, що ***додаток повинен припинити роботу***, повідомивши користувача про причину її переривання.

Представлення виключень

- ***Один спосіб:*** представлення за допомогою ***кодів помилок (error codes)***
  - Наприклад, метод може повертати true при успішному виконанні та false при виникненні винятка.
- ***Альтернативний спосіб:*** повертати 0 при успішному виконанні та ненульове ціле значення-ідентифікатор винятка.
  - Більш традиційний спосіб, проте розробник може просто ігнорувати виняток.
  - Це стало причиною появи механізму обробки винятків на основі об'єктів.
- Коли трапляється виняток, об'єкт, що його представляє, створюється в запущеному коді.
  - В об'єкті міститься інформація, яка описує оточуючий контекст для винятку.
- Потім об'єкт викидається (*thrown*) або виходить за рамки віртуальної машини в пошуку обробника (*handler*) – коду, що може обробити виняток.
  - Якщо виняток є помилкою, додаток має не постачати обробника, оскільки помилки настільки серйозні (наприклад, нестача пам'яті для віртуальної машини), що з цим практично нічого неможливо зробити.
  - Коли обробник знайдено, його код виконується, забезпечуючи обхід (workaround).
  - Інакше ВМ припиняє роботу додатку.

Обережно!

- Код, який обробляє виключення, може бути повним багів, оскільки нормально не тестується.
  - Завжди тестуйте код, що обробляє виняток.
  - Error-код типу Boolean або Integer менш осмислений, ніж назва об’єкту.
    - Наприклад, fileNotFound самодостатній, проте що може значити false?
- Також об’єкт може містити інформацію про те, що призвело до виключення.
  - Ці деталі можуть допомогти розробити доречний обхід.

Опис обробки винятків

- Для виділення блока коду, який слід захистити від винятків, використовується ключове слово try.
  - Після try-блока відразу розміщується блок catch, який задає тип винятку для обробки.

```
class Exc2 {
        public static void main(String args[]) {
                try {
                        int d = 0;
                        int a = 42 / d;
                }
                catch (ArithmeticException e) {
                        System.out.println("division by zero");
                }
        }
}
```

- ***Ще згадаємо про винятки наступній темі***

Декілька розділів catch

- У деяких випадках один і той же блок програмного коду може порушувати винятки різних типів.

```
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    }
    catch (ArithmeticException e) {
      System.out.println("div by 0: " + e);
    }
    catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("array index oob: " + e);
    }
  }
}
```

Приклад, запущений без параметрів, викликає виникнення виключної ситуації ділення на нуль

- Якщо поставимо в командному рядку один або кілька параметрів, тим самим надавши *a* значення більше нуля, наш приклад переживе оператор ділення.
  - Але в наступному операторі буде порушено виняток виходу індексу за межі масиву ArrayIndexOutOfBounds.

```
C:\> java MultiCatch
a = 0
div by 0: java.lang.ArithmeticException: / by zero
C:\> java MultiCatch 1
a = 1
array index oob: java.lang.ArrayIndexOutOfBoundsException: 42
```

Вкладені оператори try

```
class MultiNest {
  static void procedure() {
    try {
      int c[] = { 1 };
      c[42] = 99;
    }
    catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("array index oob: " + e);
    }
  }

  public static void main(String args[]) {
    try {
      int a = args.length();
      System.out.println("a = " + a);
      int b = 42 / a;
      procedure();
    }
    catch (ArithmeticException e) {
      System.out.println("div by 0: " + e);
    }
  }
}
```

Якщо в оператора try низького рівня немає розділу catch, що відповідає порушеному виключенню, стек буде розгорнуто на один щабель вище.

У пошуках доречного обробника будуть перевірені розділи catch зовнішнього оператора try.

Оператор throw

- Використовується для ручного викидання винятку.
  - Потрібно мати об’єкт підкласу класу Throwable, який можна отримати або як параметр оператору catch, або створити новий за допомогою оператору new
  - загальна форма оператора: throw Об’єктТипуThrowable;
  - throw new EmptyStackException();
- При досягненні цього оператора нормальне виконання коду негайно припиняється, а наступний за ним оператор не виконується.
  - Найближчий оточуючий блок try перевіряється на наявність відповідного йому оператора catch.
  - Якщо знайде, передасть йому управління, інакше перевіряється наступний із вкладених операторів try.
  - Так відбувається, поки не буде знайдено доречний розділ catch або обробник винятків виконуючої системи Java не зупинить програму, вивівши при цьому стан стеку викликів.

Приклад використання оператору throw

```
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    }
    catch (NullPointerException e) {
      System.out.println("caught inside demoproc");
      throw e;
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    }
    catch(NulPointerException e) {
      System.out.println("recaught: " + e);
    }
  }
}
```

```
C:\> java ThrowDemo
caught inside demoproc
recaught: java.lang.NullPointerException: demo
```

Ключове слово throws

- Якщо метод здатний порушувати винятки, які ***сам не обробляє***, він повинен оголосити про таку поведінку, щоб ***викликаючі методи могли захистити*** себе від цих винятків.
  - Для задання списку винятків, які можуть порушуватися методом, використовується ключове слово throws.
- Якщо метод у явному вигляді (оператор throw) порушує виняток відповідного класу, тип класу винятків повинен бути зазначений в операторі throws в оголошенні цього методу.
  - Попередній синтаксис визначення методу повинен бути розширений:
  - тип назва_методу(список аргументів) ***throws список_виключень*** {}

```
class ThrowsDemo {
  static void procedure() throws IllegalAccessException {
    System.out.println(" inside procedure");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try { procedure(); }
    catch (IllegalAccessException e) {
      System.out.println("caught " + e);
    }
  }
}
```

```
C:\> java ThrowsDemo
inside procedure
caught java.lang.IllegalAccessException: demo
```

Ключове слово finally

- Іноді потрібно гарантувати, що певна ділянка коду буде виконуватися незалежно від того, які винятки були порушені та перехоплені.
  - Використовується ключове слово finally.
  - Навіть тоді, коли в методі немає відповідного порушеному винятку розділу catch, блок finally буде виконаний до того, як управління перейде до операторів, що слідують за розділом try.
- У кожного розділу try повинен бути ***принаймні або один розділ catch, або блок finally***.
  - Блок finally дуже зручний для закриття файлів і звільнення будь-яких інших ресурсів, захоплених для тимчасового використання на початку виконання методу.

Приклад класу з двома методами, завершення яких відбувається з різних причин

```
class FinallyDemo {
  static void procA() {
    try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    }
    finally { System.out.println("procA's finally"); }
  }
  static void procB() {
    try { System.out.println("inside procB"); return; }
    finally { System.out.println("procB's finally"); }
  }
  public static void main(String args[]) {
    try { procA(); }
    catch (Exception e) {}
    procB();
  }
}
```

- в обох перед виходом виконується код розділу finally.

```
C:\> java FinallyDemo
inside procA
procA's finally
inside procB
procB's finally
```

Додаткові факти про роботу з операторами throws та throw

- Можна дописувати оператор throws до конструктора та викидати винятки з конструктора.
- Результуючий об’єкт не буде створено:
  - При викиді виключення з методу main() віртуальна машина перериває роботу додатку та викликає метод printStackTrace() виключення.
    - Щоб вивести на консоль послідовність вкладених викликів методів, які очікували завершення своєї роботи у момент викидання виключення.
  - Якщо метод суперкласу оголошує пункт throws, переозначений метод підкласу може не оголошувати його.
  - Нехай матимемо метод суперкласу void foo() throws IOException {},
    - переозначений метод підкласу може оголошуватись як void foo() {}, void foo() throws IOException {}, або void foo() throws FileNotFoundException {};
    - Клас java.io.FileNotFoundException субкласує IOException.

Знову про обробку винятків

```
try
{
   int x = 1 / 0;
}
catch (ArithmeticException ae)
{
   System.out.println("attempt to divide by zero");
}
```

- Коли виконання програми заходить у блок try, при спробі ділення на о віртуальна машина відповість інстанціюванням ArithmeticException та його викидом.
- Далі вона знайде catch-блок, що здатен обробити викинуті об’єкти ArithmeticException, а потім перенесе виконання в цей блок.
  - Оскільки ArithmeticException – приклад unchecked-виключення, які представляють помилки кодингу, що потрібно виправити.
  - Зазвичай їх не перехоплюють, як у прикладі.
  - Замість цього програміст вирішує проблему, що призвела до цього виключення.

```
import java.io.FileNotFoundException;
import java.io.IOException;

import media.InvalidMediaFormatException;
import media.Media;

public class Converter
{
   public static void main(String[] args)
   {
      if (args.length != 2)
      {
         System.err.println("usage: java Converter srcfile dstfile");
         return;
      }
      try
      {
         Media.convert(args[0], args[1]);
      }
      catch (InvalidMediaFormatException imfe)
      {
         System.out.println("Unable to convert " + args[0] + " to " + args[1]);
         System.out.println("Expecting " + args[0] + " to conform to " +
                            imfe.getExpectedFormat() + " format.");
         System.out.println("However, " + args[0] + " conformed to " +
                            imfe.getExistingFormat() + " format.");
      }
      catch (FileNotFoundException fnfe)
      {
      }
      catch (IOException ioe)
      {
      }
   }
}
```

Обробка кількох винятків

- Виклик методу Media.convert () у try-блоці, оскількі ВІН здатний викидати екземпляр checked-винятків InvalidMediaFormatException, IOException чи FileNotFoundException.
  - Checked-винятки повинні оброблятись або оголошуватись для викидання за допомогою оператора throws, приєднаного до методу.

Результати виводу

```
Unable to convert A to B
Expecting A to conform to RM format.
However, A conformed to WAVE format.
```

- Порожні catch-блоки FileNotFoundException та IOException ілюструють досить поширену проблему.
  - Не створюйте порожні catch-блоки без потреби: вони перекривають виключення без визначення та вирішення проблеми.
- Компілятор повідомляє про помилку, коли задається кілька catch-блоків з однаковим типом параметру після try-блоку.
  - Наприклад: try {} catch (IOException ioe1) {} catch (IOException ioe2) {}.
  - Необхідно об'єднати ці блоки в один.
- Писати catch-блоки можна в довільному порядку, проте компілятор restricts this order, коли параметр одного catch-блоку є супертипом для параметру іншого catch-блоку.
  - Catch-блок з параметром-підтипом повинен передувати catch-блоку з параметром-супертипом;
  - Інакше catch-блок для підтипу ніколи не виконається (екземпляр FileNotFoundException є також екземпляром його суперкласу IOException).
  - Наприклад, catch-блок для FileNotFoundException ПОВИНЕН передувати catch-блоку IOException.

Виконання очистки

- У деяких випадках буде потрібно виконати очистку (cleanup) коду до того, як потік виконання покине метод через викидання виключення.
  - Наприклад, потрібно закрити раніше відкритий файл, в який через нестачу місця неможливо здійснити запис.
- Для цього Java постачає блок finally.
  - Він слідує за блоками try або catch
- n the former case, виняток може оброблятись (і, можливо, перевикидатись) до виконання виконання блоку finally.
- In the latter case, виняток обробляється (і, можливо, перевикидається) після виконання блоку finally.

Очистка шляхом закриття файлів після обробки викинутого винятку

```
import java.io.IOException;

public class Copy
{
   public static void main(String[] args)
   {
      if (args.length != 2)
      {
         System.err.println("usage: java Copy srcFile dstFile");
         return;
      }

      int fileHandleSrc = 0;
      int fileHandleDst = 1;
      try
      {
         fileHandleSrc = open(args[0]);
         fileHandleDst = create(args[1]);
         copy(fileHandleSrc, fileHandleDst);
      }
      catch (IOException ioe)
      {
         System.err.println("I/O error: " + ioe.getMessage());
         return;
      }
      finally
      {
         close(fileHandleSrc);
         close(fileHandleDst);
      }
   }
```

- Лістинг моделює копіювання байтів з джерела до цільового файлу.

```
   static int open(String filename)
   {
      return 1; // Assume that filename is mapped to integer.
   }

   static int create(String filename)
   {
      return 2; // Assume that filename is mapped to integer.
   }

   static void close(int fileHandle)
   {
      System.out.println("closing file: " + fileHandle);
   }

   static void copy(int fileHandleSrc, int fileHandleDst) throws IOException
   {
      System.out.println("copying file " + fileHandleSrc + " to file " +
                         fileHandleDst);
      if (Math.random() < 0.5)
         throw new IOException("unable to copy file");
   }
}
```

Результати виводу

Якщо немає проблем

```
copying file 1 to file 2
closing file: 1
closing file: 2
```

Якщо проблеми виникли

```
copying file 1 to file 2
I/O error: unable to copy file
closing file: 1
closing file: 2
```

Незалежно від виникнення помилки вводу-виводу, finally-блок виконується наприкінці. Навіть, якщо в catch-блоці буде оператор return.

Очистка шляхом закриття файлів до обробки викинутого винятку

- Лістинг майже ідентичний попередньому.
  - Відмінність: пункт throws приєднаний до заголовка методу main() та видалений з catch-блоку.

```
import java.io.IOException;

public class Copy
{
   public static void main(String[] args) throws IOException
   {
      if (args.length != 2)
      {
         System.err.println("usage: java Copy srcFile dstFile");
         return;
      }
      int fileHandleSrc = 0;
      int fileHandleDst = 1;
      try
      {
         fileHandleSrc = open(args[0]);
         fileHandleDst = create(args[1]);
         copy(fileHandleSrc, fileHandleDst);
      }
      finally
      {
         close(fileHandleSrc);
         close(fileHandleDst);
      }
   }
```

```
   static int open(String filename)
   {
      return 1; // Assume that filename is mapped to integer.
   }

   static int create(String filename)
   {
      return 2; // Assume that filename is mapped to integer.
   }

   static void close(int fileHandle)
   {
      System.out.println("closing file: " + fileHandle);
   }

   static void copy(int fileHandleSrc, int fileHandleDst) throws IOException
   {
      System.out.println("copying file " + fileHandleSrc + " to file " +
                         fileHandleDst);
      if (Math.random() < 0.5)
         throw new IOException("unable to copy file");
   }
}
```

Результати виконання

- Цього разу обробник виключень Java виконує executes printStackTrace(), і ми спостерігаємо подібний вивід:

```
copying file 1 to file 2
closing file: 1
closing file: 2
Exception in thread "main" java.io.IOException: unable to copy file
        at Copy.copy(Copy.java:48)
        at Copy.main(Copy.java:19)
```

Твердження та їх оголошення

- ***Твердження*** *(assertion)* – оператор мови програмування, який дозволяє виражати припущення про коректність роботи програми за допомогою булевого виразу
  - Коментарі марні для запобігання помилок, оскільки компілятор їх ігнорує.
  - Твердження дозволяють розробнику закодувати припущення щодо коректності роботи додатку.
  - Якщо вираз набуває значення true, виконання продовжується до наступного твердження.
  - Інакше викидається помилка, додаток перериває роботу з діагностичним повідомленням про причину падіння.

- Існує дві форми оператора твердження (assertion statement), які починаються із зарезервованого слова assert:
  - assert *expression1*; // *expression1 – булевий вираз*
  - assert *expression1* : *expression2* ; // *expression2 – будь-який вираз, що повертає значення (але не void)*

- Коли *expression1* набуває значення false, оператор створює екземпляр класу java.lang.AssertionError.
  - Даний варіант оператору викликає безаргументний конструктор цього класу, який не пов'язує повідомлення про збій з екземпляром AssertionError.

Оголошення тверджень

```
public class AssertionDemo
{
   public static void main(String[] args)
   {
      int x = 1;
      assert x == 0;
   }
}
```

Якщо твердження ввімкнені, виведеться:

- `Exception in thread "main" java.lang.AssertionError at AssertionDemo.main(AssertionDemo.java:6)`

```
public class AssertionDemo
{
   public static void main(String[] args)
   {
      int x = 1;
      assert x == 0: x;
   }
}
```

- Аналогічно до попереднього прикладу, виведеться:
  - `Exception in thread "main" java.lang.AssertionError: 1 at AssertionDemo.main(AssertionDemo.java:6)`
  - Значення `x` додається в кінець першого рядка виводу, проте він досить загадковий.
  - Щоб зробити вивід більш змістовним, можна задати вираз на зразок `assert x == 0: "x = " + x;`.

Ввімкнення та вимкнення тверджень

- Компілятор записує твердження в class-файл.
  - Проте вони вимкнені при runtime, оскільки можуть впливати на продуктивність.
  - Твердження може викликати метод, який певний час працюватиме, впливаючи на швидкодію працюючого додатку.
- До того, як тестувати припущення про поведінку класів, потрібно ввімкнути твердження для class-файлу.
  - Виконується шляхом прописування опції (ключа) командного рядка -ea або –enableassertions, коли працює java application launcher tool.
- Ці опції дозволяють увімкнути твердження різної гранулярності на основі одного з наступних аргументів:
  - *Безаргументний сценарій*: твердження ввімкнені в усіх класах, крім системних класів.
  - *PackageName*...: твердження ввімкнені в заданому пакеті та підпакетах, задаючи назву пакету, після якої пишеться ....
  - ...: твердження ввімкнені в безіменному пакеті, який відповідає поточній директорії.
  - *ClassName*: твердження ввімкнені в класі, задаючи його назву.
- Наприклад, ввімкнути всі твердження, крім системних, при роботі додатку MergeArrays:
  - **`java –ea MergeArrays`**.

Вимкнення тверджень

- Твердження можна вимкнути з різною деталізацією (granularities): задавати опцію –disableassertions або –da.
  - Ці опції приймають ті ж аргументи, що і -enableassertions та -ea.
  - Наприклад, java -ea –da:loneclass mainclass вмикає всі твердження, крім тих, що в loneclass.
- Попередні опції застосовуються до всіх завантажувачів класів (classloaders).
  - Крім безаргументного сценарію: тоді застосування поширюється на системні класи.
  - Для ввімкнення системних тверджень, задайте -enablesystemassertions або –esa.
  - Наприклад, java -esa –ea:logging TestLogger.
  - Для вимкнення задайте прапорець -disablesystemassertions або -dsa.

Ситуації використання тверджень відносяться до категорій

- ***Ситуації стосовно внутрішнього інваріанту*** *(internal invariant)*,
  - Інваріант – дещо в коді, що має залишатись незмінним.
- ***Ситуації стосовно управляючого потоку*** *(control-flow invariant);*
- *design-by-contract*.

- Внутрішній інваріант є орієнтованою на вирази поведінкою, зміни якої не очікуються.
  - У лістингу представлено внутрішній інваріант у вигляді ланцюга операторів if-else, який виводить стан води на основі її температури.

```
public class IIDemo
{
   public static void main(String[] args)
   {
      double temperature = 50.0; // Celsius
      if (temperature < 0.0)
         System.out.println("water has solidified");
      else
      if (temperature >= 100.0)
         System.out.println("water is boiling into a gas");
      else
      {
         // temperature > 0.0 and temperature < 100.0
         assert(temperature > 0.0 && temperature < 100.0): temperature;
         System.out.println("water is remaining in its liquid state");
      }
   }
}
```

Інший забагований внутрішній інваріант

```
public class IIDemo
{
   final static int NORTH = 0;
   final static int SOUTH = 1;
   final static int EAST = 2;
   final static int WEST = 3;

   public static void main(String[] args)
   {
      int direction = (int) (Math.random() * 5);
      switch (direction)
      {
         case NORTH: System.out.println("travelling north"); break;
         case SOUTH: System.out.println("travelling south"); break;
         case EAST : System.out.println("travelling east"); break;
         case WEST : System.out.println("travelling west"); break;
         default   : assert false;
      }
   }
}
```

- Даний внутрішній інваріант записано в операторі switch з відсутнім пунктом default.
  - Випадок за умовчанням уникається, оскільки розробник вважає, що всі напрямки покриті (covered).
  - Проте це не завжди так: (int) (Math.random() * 5) може повернути 4, привівши до виконання `assert false;`, що викине AssertionError.
- Коли твердження відключені, assert false; не виконується, а баг не знайдено.
  - Щоб завжди знаходити баг, замініть на
  - throw new AssertionError(direction);

Інваріанти потоку виконання (Control-Flow Invariants)

```
public class CFDemo
{
   final static int NORTH = 0;
   final static int SOUTH = 1;
   final static int EAST = 2;
   final static int WEST = 3;

   public static void main(String[] args)
   {
      int direction = (int) (Math.random() * 4);
      switch (direction)
      {
         case NORTH: System.out.println("travelling north"); break;
         case SOUTH: System.out.println("travelling south"); break;
         case EAST : System.out.println("travelling east"); break;
         case WEST : System.out.println("travelling west");
         default   : assert false;
      }
   }
}
```

- Інваріант потоку виконання – це потік виконання, зміни якого не передбачені.
  - Пофіксимо баг попереднього лістингу, вітка default стане недоступною
  - З точки зору документування тут твердження задає код, який має ніколи не виконуватись.
  - Проте приберемо оператор break для вітки WEST, тепер перехід у вітку default можливий.

Design-by-Contract

- Підхід до проектування програмного забезпечення, що базується на:
  - Передумовах (preconditions – істинна умова при виклику методу);
  - Післяумовах (postconditions – істинна умова після успішного завершення роботи методу);
  - Інваріантах класу (class invariants – вид внутрішнього інваріанту, що застосовується до кожного екземпляру класу завжди, крім випадку переходу екземпляру з одного стану в інший).
- Вирази-твердження підтримують неформальний стиль design-by-contract розробки.

```
public class Lotto649
{
   public static void main(String[] args)
   {
      // Lotto 649 requires that six unique numbers be chosen.
      int[] selectedNumbers = new int[6];
      // Assign a unique random number from 1 to 49 (inclusive) to each slot
      // in the selectedNumbers array.
      for (int slot = 0; slot < selectedNumbers.length; slot++)
      {
           int num;
           // Obtain a random number from 1 to 49. That number becomes the
           // selected number if it has not previously been chosen.
           try_again:
           do
           {
               num = rnd(49) + 1;
               for (int i = 0; i < slot; i++)
                    if (selectedNumbers[i] == num)
                        continue try_again;
               break;
           }
           while (true);
           // Assign selected number to appropriate slot.
           selectedNumbers[slot] = num;
      }
      // Sort all selected numbers into ascending order and then print these
      // numbers.
      sort(selectedNumbers);
      for (int i = 0; i < selectedNumbers.length; i++)
           System.out.print(selectedNumbers[i] + " ");
   }
```

Передумови

- Твердження часто використовують для задоволення передумов допоміжних методів шляхом перевірки їх аргументів на легальність.

Додаток моделює Lotto 6/49

```
   static int rnd(int limit)
   {
      // This method returns a random number (actually, a pseudorandom number)
      // ranging from 0 through limit - 1 (inclusive).
      assert limit > 1: "limit = " + limit;
      return (int) (Math.random() * limit);
   }

   static void sort(int[] x)
   {
      // This method sorts the integers in the passed array into ascending
      // order.
      for (int pass = 0; pass < x.length - 1; pass++)
         for (int i = x.length - 1; i > pass; i--)
            if (x[i] < x[pass])
            {
               int temp = x[i];
               x[i] = x[pass];
               x[pass] = temp;
            }
   }
}
```

- Допоміжний метод rnd() повертає випадково обране ціле число між 0 та limit-1.
- Твердження перевіряє передумову, що значення limit має бути 2 або більше.

```
public class MergeArrays
{
   public static void main(String[] args)
   {
      int[] x = { 1, 2, 3, 4, 5 };
      int[] y = { 1, 2, 7, 9 };
      int[] result = merge(x, y);
      for (int i = 0; i < result.length; i++)
         System.out.println(result[i]);
   }

   static int[] merge(int[] a, int[] b)
   {
      if (a == null)
         throw new NullPointerException("a is null");
      if (b == null)
         throw new NullPointerException("b is null");
      int[] result = new int[a.length + b.length];
      // Precondition
      assert result.length == a.length + b.length: "length mismatch";
      for (int i = 0; i < a.length; i++)
         result[i] = a[i];
      for (int i = 0; i < b.length; i++)
         result[a.length + i - 1] = b[i];
      // Postcondition
      assert containsAll(result, a, b): "value missing from array";
      return result;
   }
```

Постумови (Postconditions)

- Твердження часто використовуються для задоволення постумов допоміжного методу шляхом перевірки результату роботи на легальність.

Тут твердження перевіряє постумову: всі значення з двох масивів, що зливаються, присутні в результуючому масиві.

Продовження лістингу

```
static boolean containsAll(int[] result, int[] a, int[] b)
{
   for (int i = 0; i < a.length; i++)
      if (!contains(result, a[i]))
         return false;
   for (int i = 0; i < b.length; i++)
      if (!contains(result, b[i]))
         return false;
   return true;
}
```

```
static boolean contains(int[] a, int val)
{
   for (int i = 0; i < a.length; i++)
      if (a[i] == val)
         return true;
   return false;
}
}
```

- Постумова не задовольняється через баг у коді.
- Окрема передумова перевіряє, щоб злитий масив по довжині відповідав сумі довжин масивів, що зливаються.

Інваріанти класу. Приклад

- Нехай екземпляри класу містять масиви, елементи яких відсортовані в зростаючому порядку.
  - Можна включити в клас метод isSorted(), який повертає true, коли масив все ще відсортований.
  - При перевірці того, що при виході з конструктору чи методу задовольняється припущення про відсортованість масиву, задають `assert isSorted();` перед виходом.

ДЯКУЮ ЗА УВАГУ!

Наступна тема: інкапсуляція та взаємодія класів у Java-додатках

Уникнення тверджень

- Не слід використовувати твердження для перевірки аргументів, які передаються публічним методам. Причини:
  - Їх перевірка є частиною контракту, що існує між методом, та тим, хто його викликає.
    - Якщо перевіряти такі аргументи з вимкненими твердженнями, контракт порушується, оскільки аргументи не будуть перевірятись.
  - Твердження також не дають викидатись відповідним виключенням.
    - Наприклад, коли в публічний метод передається нелегальний аргумент, часто викидається java.lang.IllegalArgumentException або java.lang.NullPointerException.
    - Проте замість них викинеться AssertionError.
- Також уникайте використання тверджень для виконання роботи required by the application to function correctly.
  - This work is often performed as a side effect of the assertion's Boolean expression.

Уникнення тверджень

- Коли твердження вимкнені, робота не виконується.
  - Наприклад, For example, suppose you have a list of Employee objects and a few null references that are also stored in this list, and you want to remove all of the null references.
  - It would not be correct to remove these references via the following assertion statement:

  assert employees.removeAll(null);

- Хоч оператор твердження не викидатиме AssertionError, оскільки існує принаймні одне null-посилання в employees list, додаток, який залежить від виконання цього оператору буде fail, якщо твердження будуть вимкнені.

- Instead of depending on the former code to remove the null references, you would be better off using code similar to the following:
  boolean allNullsRemoved = employees.removeAll(null);
  assert allNullsRemoved;
  - Цього разу всі null-посилання видаляються, незалежно від того, ввімкнені чи ні твердження, а Ви досі можете задавати твердження для перевірки того, що nulls were removed.