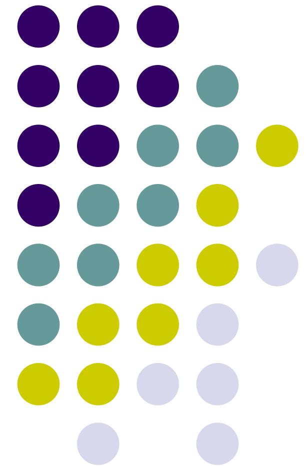
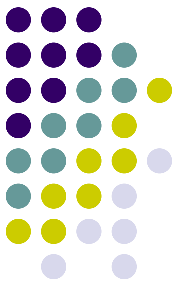
Мова програмування Java та технології J2EE Модуль “Мова програмування Java”

Лекція 4. Виключні ситуації.

Узагальнене програмування на мові Java (Generics).



Виключні ситуації



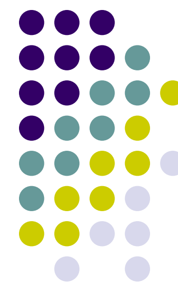
Виключна ситуація (виключення) – це помилка часу виконання.

- Приклади: недостатньо ресурсів, ділення на 0, “запрошений файл не існує” тощо

Як обробляти такі помилки?

2. Спробувати запобігти їм, застосовуючи нагромадження перевірок if-then-else. Недоліки підходу:
    - при виконанні перевірок може виникнути помилка часу виконання (наприклад, «недостатньо пам’яті»)
    - безліч перевірок значно ускладнює програмний код
    - ці перевірки вже зроблено в тих функціях/операторах, які застосовуються
3. **Застосувати механізм обробки виключних ситуацій**.

Обробка виключних ситуацій



Обробка виключних ситуацій – це механізм, який застосовується для обробки помилок та описує реакцію програми на помилки часу виконання. Дозволяє логічно відділити код прикладної програми від обробки помилок.

Переваги підходу:

- Спрощення програмного коду
- Запобігання зайвих перевірок - не повторюються перевірки, реалізовані в функціях, що викликаються

Механізм обробки виключних ситуацій реалізується JVM та полягає у кроках:

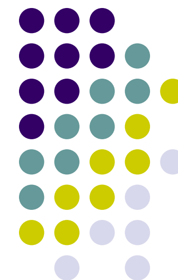
1. Після генерації виключення у коді програми JVM перериває нормальне виконання та передає управління блоку обробки виключення
2. Після успішної обробки виключенні виконання продовжується з блоку, який слідує за блоком обробки виключення

При розробці програм програмісту окрім використання обробки виключних ситуацій може знадобитися самому **ініціювати виключні ситуації**.

Ініціювати виключну ситуації необхідно, якщо є порушення семантичних обмежень програми:

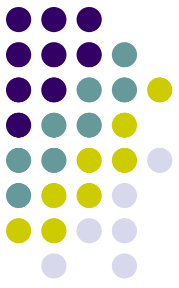
- неочікувані аргументи тощо
- наприклад, JVM генерує виключення при діленні на 0, створенні масиву від'ємної розмірності, якщо не вистачає пам'яті для створення об'єкту і т.д.

Синтаксис обробки виключень



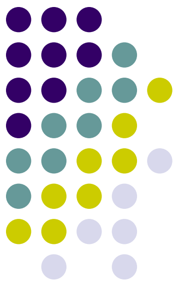
```
try {
    …                           // код, що контролюється
}
catch (ТипВиключення1 змінна1) {
    …                       // перехват виключення
}
catch (ТипВиключення2 змінна2) {
    …                           // перехват виключення
}
catch (ТипВиключенняN зміннаN) {
    …                           // перехват виключення
}
finally{
    …                     // код гарантованого виконання
}
… // З цього рядка продовжується виконання програми
   після успішного перехвату виключення
```

Приклад блоку обробки виключень



```
int i;
double div;
double dividor=0;
try {
    i=Integer.parseInt("абракадабра");
    div=i/dividor;
}
catch(NumberFormatException e){
    System.out.println("Це не число");
}
catch(ArithmeticException e){
    System.out.println("Ділення на ноль");
}
finally {
    System.out.println("Блок гарантованого виконання");
}
```

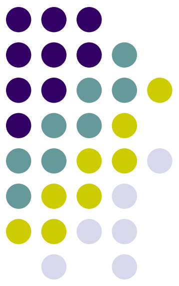
Синтаксис обробки виключень. Продовження



- Блок обробки виключень:
  - `try` – секція коду, що контролюється
  - `catch` – секції перехвату виключення
    - Можливі декілька секцій `catch` для перехвату виключення в залежності від його типу
    - Спрацьовує тільки один `catch`, перший з тих, що відповідає типу виключення
  - `finally` – секція гарантованого завершення
    - Зазвичай використовується для очистки ресурсів
    - Виконується завжди, навіть у наступних випадках:
      - якщо не була ініційована виключна ситуація
      - якщо не була "спіймана" виключна ситуація

- Порядок виконання: `1) try – 2) catch – 3) finally`
- Можливі варіанти використання секцій
  - `try-catch-finally`
  - `try-catch`
  - `try-finally`

Приклад застосування секції гарантованого завершення

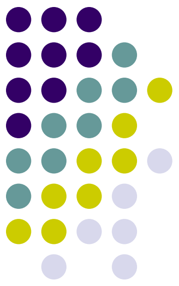


У наступному прикладі наведено, що блок finally виконується, навіть якщо виключення не “спіймане”

```
public void foo () {
     int i;
     try{
          i=Integer.parseInt("абракадабра");
     }
     finally {
          System.out.println("Гарантоване завершення");
     }
}
```

foo видає: Гарантоване завершення

Запитання із співбесід

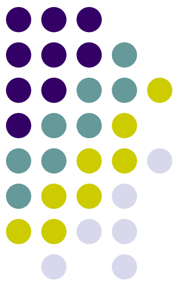


- На Вашу думку, коли в коді на мові Java не спрацює секція finally?

```
try {
    System.out.println("Hello from try");
    System.exit(0);
}
finally {
    System.out.println("Hello from finally");
}
```

- Програма видасть тільки: Hello from try
- Поясність чому

Ініціація виключень



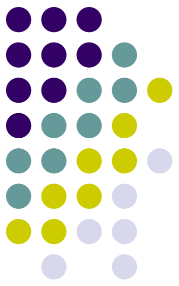
- **`throw`** – оператор для ініціації виключень

- Виключення в Java – це об’єкти, успадковані від класу **`Throwable`**
- Класи виключень
  - Ієрархія класів виключень J2SE: `Throwable`, `Exception`, `Error`(див. далі)
  - JavaSE пропонує корисні для використання класи, наприклад: `IllegalArgumentException`, `IllegalStateException`

  - Власні класи виключень зазвичай успадковують від **`Exception`**

```
class TestException extends Exception {
    TestException() { super(); }
    TestException(String s) { super(s); }
}
```

Ініціація виключень. Продовження



- Ініціація нового об’єкту виключення –
  об’єкт-виключення необхідно створити, «**new**»

```
public void foo (int a) {
    if (a < 0)
        throw new IllegalArgumentException();
    ...
}
```

  Зверніть увагу на застосування `IllegalArgumentException` для перевірки параметрів

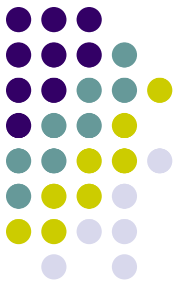
- Повторна ініціація існуючого об’єкту виключення

```
int i;
try {
    i=Integer.parseInt(s);
}
catch(NumberFormatException e){
    System.out.println("Це не число");
    throw e;
}
```

- Виключення можуть утворювати “люнцюги”

```
try {
    ...;
} catch (LowLevelException le) {
    le.printStackTrace();
    throw new HighLevelException(le);   // ланцюг виключень
}
```

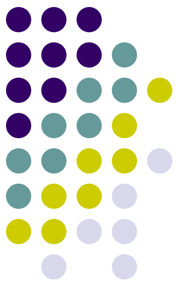
Конструктори виключень



- При розробці власного класу виключень доцільно реалізувати конструктор із наступними сигнатурами:

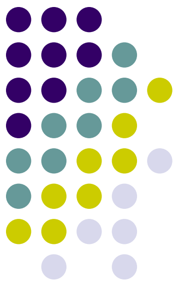
```
class TestException extends Exception {
    TestException() { super(); }
    TestException(String s) { super(s); }
    TestException(String s, Throwable ex) { super(s, ex); }
    TestException(Throwable ex) { super(ex); }
}
```

Виключення та JVM



- **`throw`** та JVM
  - Стек JVM thread:
    - JVM перериває управління по стеку викликів у потоці виконання (JVM thread) до тих пір, поки не знайде блок обробки даного виключення. Якщо такий блок не знайдено, то викликається метод ThreadGroup.uncaughtException
  - **`synchronized`**
    - Механізм обробки виключень інтегровано із моделлю конкурентного доступу до ресурсів
    - Звільняються захвачені ресурси

Декларація методу, який може ініціювати виключну ситуацію



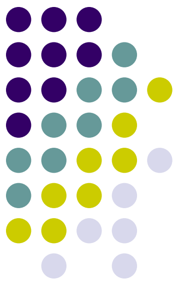
- **throws** – ключове слово у декларації методу/конструктору для зазначення типів виключень, що ініціюються

```
Модифікатори Тип Назва (Параметри) throws ТипВиключ1,
   ТипВиключ2,…,ТипВиключN {
        …
}
```

- Приклад:

```
public void createFile(String name)throws IOException {
       …
}
```

Контроль виключень компілятором



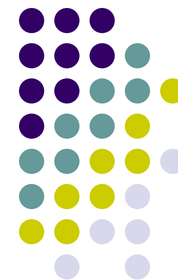
- **Java-програми стійкі до помилок**
- Обробка виключних ситуацій в Java є обов'язковою для програміста та контролюється компілятором

- 2 типи виключень:
  - виключення, обробка яких **перевіряється** компілятором
    - **ckecked exceptions**
  - виключення, обробка яких **не перевіряється** компілятором
    - **unckecked exceptions**

- Ці типи виключені базуються на ієрархії виключень в Java

- Якщо в методі ініціюється виключення, що перевіряється компілятором, то повинно бути виконане одне з наступних правил:
  - **Виключення повинно бути оброблене в тілі методу**
    - обрамити операторами `try-catch-finally` той блок команд, в якому ініціюється виключення
  - **Метод повинен делегувати обробку виключення**
    - Застосувати ключове слово `throws` в декларації методу/конструктору та вказати класи виключень, що ініціюються

Приклади



- Невірно:

```
public void createFile(String name) {
  File f;
  f=new File(name);
  f.createNewFile(); // помилка компіляції
}
```

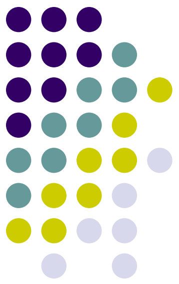
- Вірно:

```
public void createFile(String name)  {
    File f;
    f=new File(name);
    try {
        f.createNewFile(); // ok
    } catch (IOException ex) {
        ...
    }
}
```

- Вірно:

```
public void createFile(String name) throws IOException{
  File f;
  f=new File(name);
  f.createNewFile(); // ok
}
```

Ієрархія виключень

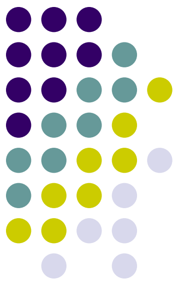


- `Throwable` – клас-пращур усіх виключень
- Тільки об'єкти класу `Throwable` (або нащадків) можуть наступне:
  - ініціювати виключення за допомогою оператора `throw`
  - бути аргументами секції `catch`

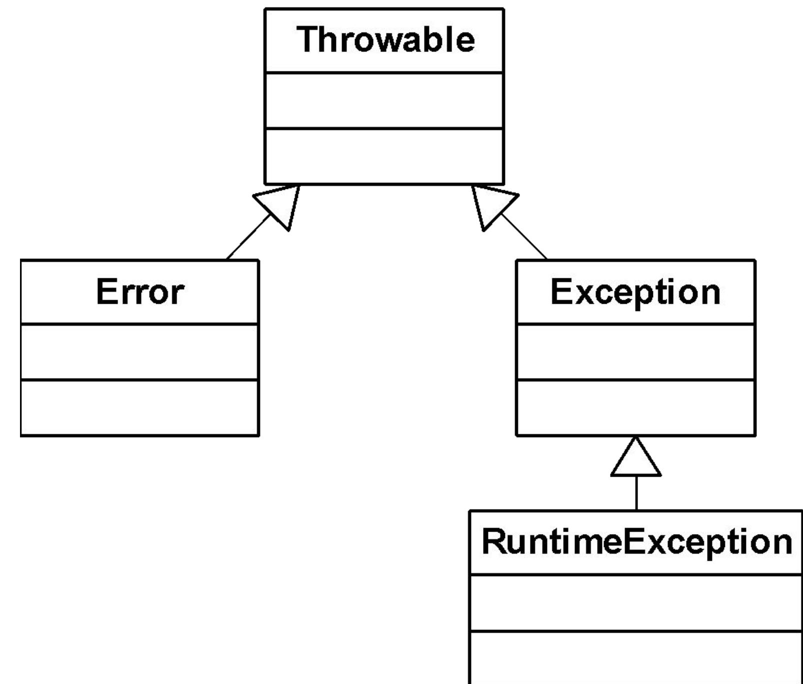
Throwable
+getStackTrace() +printStackTrace() +getCause() +getMessage() +getLocalizedMessage()

- `Throwable` містить корисні функції
  - "знімок" стеку викликів, які ініціювали виключення:
    - `printStackTrace` – друк "знімку" стеку у консоль
    - `getStackTrace` – отримати "знімок" стеку для обробки
  - ланцюг виключень:
    - `getCause` – отримати виключення, яке спричинило поточне
  - текст повідомлення
    - `getMessage` – отримати текст опису помилки
    - `getLocalizedMessage` – отримати локалізований текст опису помилки

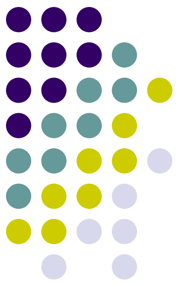
Ієрархія виключень. Продовження



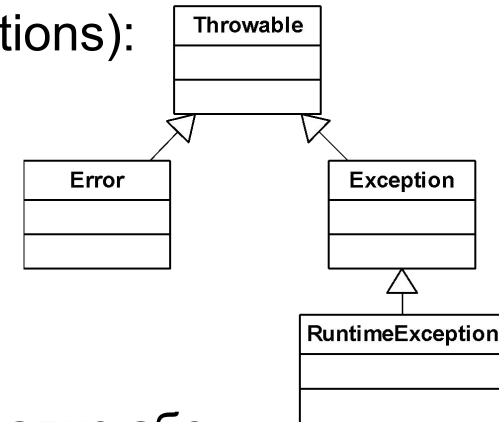
- Головні класи у ієрархії виключень:
  - **`Error`** – серйозна помилка, після якої майже неможливо відновити роботу програми
    - Наприклад, `OutOfMemoryError`, `NoClassDefFoundError`
  - **`Exception`** – виключення, яке необхідно обробити
    - Наприклад, `IOException` (помилка вводу/виводу), `SQLException`, `CertificateException`, `PrintException`
  - **`RuntimeException`** – виключення, обробка яких не впливає значним чином на коректність роботи програми
    - Наприклад, `NullPointerException`, `IndexOutOfBoundsException`, `IllegalArgumentException`, `ClassCastException`



Ієрархія виключень. Контроль виключень компілятором

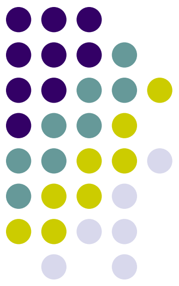


- **Не перевіряються** компілятором (unckecked exceptions):
  - `Error` та нащадки
  - `RuntimeException` та нащадки
- **Перевіряються** компілятором (checked exception):
  - Усі інші



- **Чому не перевіряються** виключення типу `Error`?
  - Тому що відновлення після таких помилок дуже складне або неможливе
  - Наприклад, після помилки "клас не знайдено" (NoClassDefFoundError) відновитися неможливо.
- **Чому не перевіряються** виключення типу `RuntimeException`?
  - Тому що обробка таких помилок не впливає значним чином на коректність роботи програми
  - Виникнення таких виключень у 90% випадків свідчить про те, що необхідно доопрацювати код програми. Наприклад, виникнення NullPointerException означає, що програміст не перевірив посилання ("вказівник на об'єкт") перед зверненням до його поля/методу.

Виключення та перекриття/приховування методів



- Пригадаємо тему “ООП у мові Java” та слайд “Правила перекриття/приховування методів”:
- Правила щодо секції **throws** методу, що приховує/перекриває інший метод
  - Кількість виключень не збільшується
  - Тип кожного виключення може бути уточнений (може вказуватись нащадок)

```
class Point {
    int x, y;
    void move(int dx, int dy) {…}
}
class CheckedPoint extends Point {
    void move(int dx, int dy)
                throws BadPointException {…} // error!!!
}
```

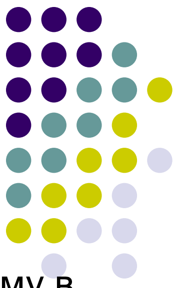
Виключення та static

- Пригадаємо тему “ООП у мові Java”, слайд “Модифікатор static”

- При ініціалізації статичних полів не допускається породження checked виключення
- У блоках статичної ініціалізації класу заборонено породження checked виключень, які виходять за рамки блоку

```
class My {
    static File file = new File("C:\\a.txt");
    static {
        file.createNewFile(); // exception
                              // compile error
    }
    …
}
```

Рекомендації по обробці виключень у реальних системах



- **Користуйтеся записом в лог-файл**
  - У реальних системах лог-файл – потужний засіб для відслідковування помилок. Тому в секції catch викликайте запис в лог-файл
  - У реальних системах у секції `catch` **не використовуйте `printStackTrace`**, а користуйтеся записом в лог-файл

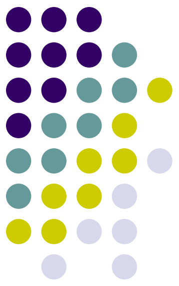
  Запис в лог буде розглянуто у подальших лекціях

- **Не “ковтайте” виключення у секції catch !!!!**
  - запишіть помилку в лог
  - ініціюйте подальшу обробку помилки
    - створіть ланцюг виключень, або
    - повторно ініціюйте перехвачене виключення

```
try {
     ...
} catch (NamingException ex) {
     Logger.getLogger(this.getClass().getName()).log(Level.SEVERE,null,
    ex);
     throw new MyException("Опис помилки", ex);
}
```

- **Намагайтеся не застосовувати єдиний catch(Exception ex)**
  - `catch (Exception ex)` буде перехватувати усі виключення – checked та unchecked.
    - Досить часто виключення типу unchecked свідчать про некоректно написану програму. Застосування єдиної секції catch (Exception ex) ускладнить пошук помилки.
  - Вказуйте у catch саме той тип виключення, який необхідно обробити
    - `catch (NamingException ex)`, `catch(SqlException ex)` тощо
  - Застосовуйте декілька секцій catch – кожний catch для свого типу виключення

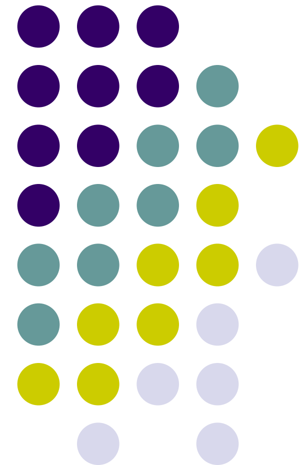
Література



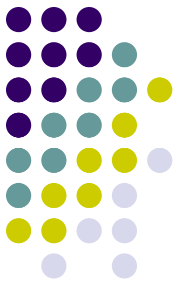
- Кей С. Хорстманн, Гари Корнелл. **Java 2. Библиотека профессионала, том 1. Основы**. - Вильямс.: 2010. - 816с.
- Брюс Эккель. **Философия Java**. – Питер, 2008. – 640 с.
- **The Java Tutorial.** http://download.oracle.com/javase/tutorial/java/TOC.html
- James Gosling, Bill Joy, Guy Steele. **The Java Language Specification**. - Addison Wesley. - 3 edition. - 2005. - 688p. -http://java.sun.com/docs/books/jls/

Лекція 4

Узагальнене програмування на мові Java (Generics)



Узагальнення (Generics)

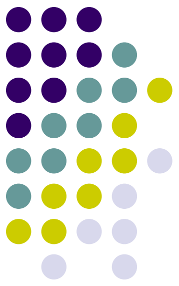


- Узагальнене програмування – це розробка коду, який може бути багаторазово використаний із об’єктами різних типів

- Основні класи задач, які потребують застосування узагальнень:
  - Розробка функцій-утиліт для колекцій (пошук, max, min, avg, sum тощо)
  - Розробка контейнерів для об’єктів різних типів (стек, колекція тощо)

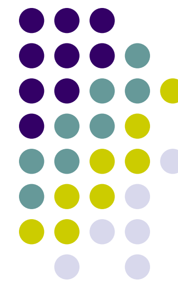
- Передумови появи узагальнень в Java
  - Java – широко застосовується, але
    - Відсутній аналог template C++
    - При програмуванні класів та алгоритмів, які потребують узагальнення, Java-код перевантажується застосуванням об’єктів типу Object та приведенням типів, насамперед, при роботі з колекціями
    - Помилки часу виконання при помилковому приведенні типів

Історія появи узагальнень в Java



- Задача - розширити систему типів мови, що широко застосовується і до якої висуваються вимоги жорсткої зворотної сумісності
- Роботу розпочато у 1999р.
- Деякі деталі із проробки задачі:
  - Специфікація “JSR-014: Adding Generics to the Java Programming Language” розроблялася протягом 1999-2004
  - Розширення системи типів підстановочними типами (wildcards) здійснено у співпраці Sun та університету м.Орхус (Данія)
    - Цікаво – один із відомих уродженців м.Орхус – Бйорн Страуструп, автор мови C++
- Узагальнення побачили світ в J2SE 5 (2004р.)

Приклад без застосування узагальнень



- Перевантаження коду змінними типу Object

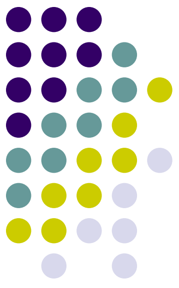
```
public class Box {
    private Object object;
    public void add(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

- Перевантаження коду приведенням типів

```
public static void main(String[] args) {
    Box integerBox = new Box(); // домовимося передавати в
                                // Box значення Integer
    integerBox.add("10"); // увага – це значення типу String
    ...
    Integer someInteger = (Integer)integerBox.get();// помилка
                                                   //часу виконання
    System.out.println(someInteger);
}
```

- Якщо негаразд із типами - помилка часу виконання
  - `java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer`

Приклад застосування узагальнень



Застосуємо узагальнення – отримаємо більш безпечний код, який краще читається

- Замість `Object` застосовуємо “типи-параметри”

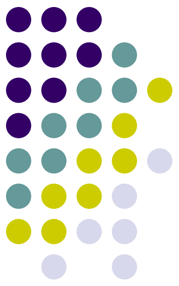
```
public class Box<T> {
    private T t;
    public void add(T t) { this.t = t; }
    public T get() { return t; }
}
```

- Не потрібно приводити типи

```
public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    integerBox.add("10");                      // Помилка компіляції
    Integer someInteger = integerBox.get(); // Не потрібне приведення
                                            // типів
    System.out.println(someInteger);
}
```

- Негаразд із типами - помилка компіляції

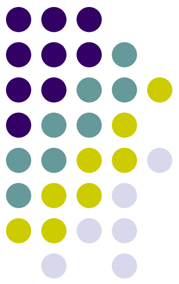
Реалізація узагальнень в Java



- Це елементи мови
- Це функціональність **компілятора**, яка дозволяє виявити певні помилки на стадії компіляції

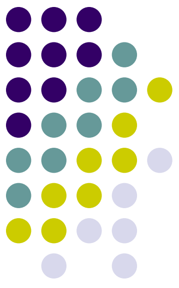
- Це **не** функціональність **JVM**
- На стадії виконання (runtime) уся інформація про узагальнення стирається
  - Через вимоги жорсткої зворотної сумісності – старий байт-код повинен працювати на нових JVM
- Узагальнення не потребують додаткових ресурсів часу виконання

Елементи мови, які узагальнюються



- Що може бути узагальнене
  - Класи
    - але не всі, див. нижче
  - Інтерфейси
  - Методи
  - Конструктори
- Які типи можуть бути параметрами для узагальнення
  - Класи
  - Інтерфейси
  - Масиви
- Які типи не можуть бути параметрами для узагальнення
  - Примітивні типи (але класи-оболонки можуть)
- Які класи не можуть бути узагальнені
  - Enum
    - Чому?
  - Клас Throwable та його нащадки
    - Обмеження викликане тим, що механізм catch у JVM не працює з параметризованими класами

Узагальнені типи



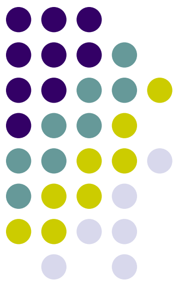
- **Узагальнені типи – узагальнені класи та узагальнені інтерфейси**
- **Узагальнений клас** – клас з принаймні однією змінною типу

```
public class Box<T> {
    private T t;
    public void add(T t) { this.t = t; }
    public T get() { return t; }
}
```

`Box` – узагальнений клас, який вводить **змінну типу** `T`
Межі дії змінної типу – весь клас

- Декілька змінних типу
  `class Suitecase<T,U> { …` - узагальнений клас з двома змінними типу
- Успадкування для узагальнених класів/інтерфейсів
  `Suitecase<T,U> extends Box<T> {…}`
- Угода щодо назв змінних типів у JavaSE
  - E – Element (використовується у Java Collections Framework)
  - K – Key
  - T – Type
  - V - Value
- Узагальнення інтерфейсів відбувається аналогічно

Узагальнені типи. Продовження



- Термін **“параметризований тип”** означає виклик узагальненого класу

```
Box<T> - узагальнений клас із змінною типу T
Box<Integer> - параметризований тип,
              із параметром (аргументом) Integer
```

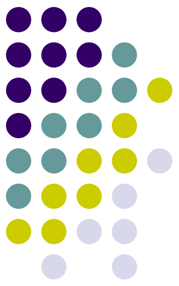
- Інші приклади параметризованих типів

```
Vector<String>
Seq<Seq<A>>
Collection<Integer>
Pair<String,String>
Iterator<int[]> - параметризація масивом
```

- Виклик конструктора параметризованого типу

```
Box<Integer> integerBox = new Box<Integer>();
```

Узагальнені методи



- Узагальнений метод – це метод з принаймні однією змінною типу

```
class Inspector {
    public <T> void inspect(T t) {
        System.out.println(t.getClass().getName());
    }
}
```

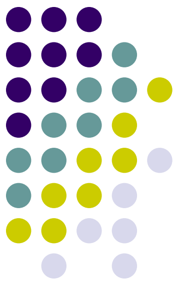
Змінна типу вказується після модифікаторів методу

- Виклик методу

```
Inspector i = new Inspector();
String s = "Hello";
i.inspect(s);               // короткий синтаксис
i.<String>inspect(s);       // повний синтаксис
```

- Узагальнення конструкторів відбувається аналогічно

Обмеження для змінних типу



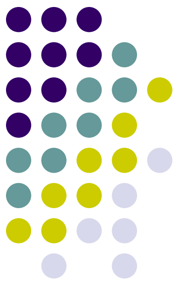
- Подібного немає в C++
- **extends &** - ключові слова
  - **extends** – означає, що параметр типу повинен успадковувати вказаний клас чи реалізовувати вказані інтерфейси
  - **&** - дозволяє вказати декілька типів, які мають бути успадковані або реалізовані (один клас, декілька інтерфейсів). “,” застосувати не можна, оскільки це роздільник між змінними типу

```
class Inspector {

    public <T extends Number&Comparable> void inspect(T t) {
    …
    }
}

Inspector i = new Inspector();
String s = "Hello";
i.inspect(s);              // помилка компіляції,
                           // оскільки s - це не Number & Comparable
```

Повторне використання коду для параметризованих класів



- Успадкування – відомий інструмент для повторного використання коду. “Працює” для узагальнень наступним чином:

```
Вірно: Integer extends Number, Double extends Number

public class Box<T> {
    T t;
    public void add(T t) { this.t = t; }
    }
}
Box<Number> box = new Box<Number>();
box.add(new Integer(10)); // OK
box.add(new Double(10.1)); // OK
```

- Як реалізувати повторне використання коду для параметризованих класів?

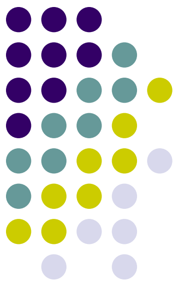
```
Невірно: Box<Integer> extends Box<Number>, Box<Double> extends Box<Number>

public void boxTest(Box<Number> n) { ...}

boxTest(new Box<Integer>()); // error
boxTest(new Box<Double>()); // error
```

- **Відповідь – підстановочні типи**

Підстановочні типи (wildcards)



- Подібного немає в C++
- Код без підстановочного типу

```
public void boxTest(Box<Number> n) { ...}

boxTest(new Box<Integer>()); // error
boxTest(new Box<Double>());  // error
```

- Код із підстановочним типом
  - **Застосування ?**

```
public void boxTest(Box<?> n) { ... } // але відсутні обмеження щодо "?"

boxTest(new Box<Integer>()); // ok
boxTest(new Box<Double>());  // ok
```

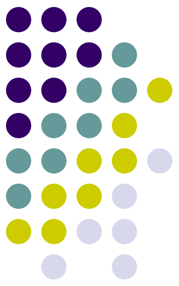
  - **Застосування ? extends Number**

```
public void boxTest(Box<? extends Number> n) { ...}

boxTest(new Box<Integer>()); // ok
boxTest(new Box<Double>());  // ok
```

- Ключові слова
  - `?` – підстановочний тип, означає "якийсь параметр узагальненого типу"
  - `extends - <? extends` *`Тип`*`>` - будь-який тип-нащадок *Тип*
  - `super  - <? super` *`Тип`*`>` - будь-який тип-пращур *Тип*
- Застосування підстановочних типів
  - Тільки в методах/конструкторах
  - Тільки для тих параметрів методів, які є узагальненими типами

Література



- Кей С. Хорстманн, Гари Корнелл. **Java 2. Библиотека профессионала, том 1. Основы**. - Вильямс.: 2010. - 816с.
- **The Java Tutorial.** http://download.oracle.com/javase/tutorial/java/TOC.html
- James Gosling, Bill Joy, Guy Steele. **The Java Language Specification**. - Addison Wesley. - 3 edition. - 2005. - 688p. -http://java.sun.com/docs/books/jls/