Мова програмування Java 2
(Java SE 6, Java SE 7)

Основні парадигми:

- Реалізовано ООП (без варіантів як у С++): підтримка об'єктів.
- Незалежність від операційного середовища.
- Багатопоточність.
- Динамічність (рефлексія).
- Безпечність.
- Інтерпретована мова (також реалізовано JIT-технологію).
- Простота (за все відповідають класи та об'єкти).

Твердження щодо Java

- Мова Java – це розширення HTML;
- Мова JavaScript – це спрощена версія Java;
- Мова Java – це ще одна мова програмування. Так у нас є C++.
- Програми на Java інтерпретуються, таким чином додатки будуть працювати досить повільно;
- Всі програми на Java виконуються під управлінням Web-броузерів.

БАЗОВІ ТИПИ ДАНИХ

Кожний об'єкт в Java повинен мати тип.

А) Цілочислові типи

- int 4 байти ОП
- short 2 байти ОП
- long 8 байтів ОП
- byte 1 байт ОП

Константи цілочислового типу:

- вісымкові константи: 0, 067, 013L. Bad: 097, 023l
- десяткові константи: 2011, 2012L. Bad: 20ff37, 2012l
- шістнадцяткові константи: 0x0f100. Bad: 0xzz97, 0X00ff, 0x0005l

Початкова ініціалізація базового типу (в подальшому для усіх базових типів)

```
int vacDays = 24;
final double CM_PER_INCH=2.54;
```

БАЗОВІ ТИПИ ДАНИХ

Б) Дійсний тип

- float 4 байти ОП
- double 8 байт ОП

Константи дійсного типу (відповідають стандарту IEEE 754):

2.5, 2.0e-4, 2.5D, 2.5F. Bad: 0x00ffe-3, (0x01ff p-3)

0x01ffp-3 мантиса - шістнадцяткова, порядок - десятковий.

В стандарті IEEE 754 також визначені наступні константи:

- нескінченний плюс: Double.NEGATIVE_INFINITY
- нескінченний мінус: Double.NEGATIVE_INFINITY
- NaN (не число): Double.NaN

В) літерний тип

char 2 байти ОП

Java SE 5.0 Кодування UTF-16 (до цього Unicode 1.0)

Г) логічний (булівський) тип

boolean

Значення: константи true та false

БАЗОВІ ТИПИ ДАНИХ

Таблиця операцій в Java:

( ) [ ] .

++ -- ~ ! +(унарний) -(унарний) new

* / %

+ -

>> >>> <<

< <= > >= instanceof

== !=

&

^

|

&&

||

?:

= += -= *= /= |= `= <<= >>= >>>=

Д) Перерахований тип:

enum Color {BLACK, GREEN, WHITE};

Color col1=BLACK; // OK

Color col2=null; // BAD

Перетворення типів операндів у виразах

По аналогії з мовою програмування С++ в мові JAVA передбачено як явне, так і неявне перетворення тирів даних при обчисленні виразів.

Неявне перетворення типу операнду:

- в унарних операціях, таких як ++ та --, операнди типу byte та short перетворюються в int;
- для бінарних операцій, якщо в обчисленні приймає участь операнд типу long, то другий теж перетворюється в long, інакше обидва операнди перетворюються в int;
- для бінарних операцій, якщо в обчисленні приймає участь один операнд double, то другий теж перетворюється в double, в противному випадку обидва операнди перетворюються в float.

Явне перетворення типу операнду:

Унарна операція перетворення типу виразу (змінної) записується як назва типу в дужках. За пріоритетом ця унарна операція має найвищий пріоритет.

Відмітимо:

- булівський тип не можна перетворити в інший тип,
- тип об'єкту можна перетворити в родовий клас.

Рядкові дані

Ж) Рядкові константи та рядкові змінні

"кібернетик"

```
String s1="кібернетик";
String s2="кібернетик "+ 3 + "курс"; // операція + для рідків, що стоять зліва
System .out . println (s2+ 3.e-4);
```

В загальному випадку:

""+<екземпляр класу>

// еквівалентно: ""+<екземпляр класу>.toString()

Важливо:

Незмінність рядків: одна з парадигм Java.

Класи String та StringBuilder

Клас String

Конструктори:

```
String ()
String(<рядкова константа>)
String(String copy)
```

Клас StringBuilder

Використовується для редагування рядків по місту (тобто самого рядка)

Технологія використання:

```
StringBuilder temp = new StringBuilder();
// редагуємо текст
String s4=temp.toString();
```

Метод equals

```
String s1="кібернетик";
String s2="кібернетик";
    if (s1= =s2) // результат завжди false
    if (s1.equals(s2))_// результат в даному випадку true
    if (s1.compareTo(s2))_// порівняння в стилі C++: якщо рівні, то 1 …
```

“Перша” програма на Java

```
import java.lang.String;
import java.util.Arrays;
public class Welcome {
public static void main (String []args){
      String [] greeting=new String[3];
      greeting[0] = "Welcome  to Eclipse ";
      greeting[1]= " Version 3.7-win32";
      greeting[2]= new String (greeting[1]);// копіюємо так
      greeting[2]= greeting[1].toString(); // копіюємо так
//The method clone() from the type Object is not visible
//String не реалізує інтерфейс clone()
// BAD:     greeting[2]=(String)greeting[1].clone();
//greeting[3]=null;
      double d1=3.66;
      for (String ss: greeting)
        System.out.println(ss);
      arrayExample();  //робота з масивами
      commandLine(args); //обробка командного рядка
      multiDimArrays(); //”зубчасті” масиви
}
```

Масиви

```
import java.lang.String;
import java.util.Arrays;
// робота з масивами
public static void arrayExample() {
// масиви інтерпретуються як об'єктні типи
    int [   ] array1;  //посилка на масив array1=null
      array1= new int[10]; // початкові значення нуль
      array1[0]=25;
    for (int iter=0; iter < array1.length; iter++ )
      System.out.println("array1[" +iter+ "]=" +array1[iter]);
//початкова ініціалізація без new
    int[]array2={1,2,3,4,5};
//початкова ініціалізація з new
    int []array3=new int[]{6,7,8,9,10};
for (int iter=0; iter < array3.length; iter++ )
System.out.println("array3[" +iter+ "]=" +array3[iter]);
//копіювання масивів
int []array4= Arrays.copyOf(array3, array3.length*2);
}
```

Робота з даними командного рядка

```
public static void commandLine(String cmdLine[]) {
    if (cmdLine.length==0) {System.out.println("Cmd Line is empty");}
    for (String ss: cmdLine)
     System.out.println(ss);
}
```

Робота з багатомірними масивами

```
public static void multiDimArrays() {
    int [][]mult = new int [5][10];
    int [][]mult1 = {//приклад "хитрого" масиву
            {1,2,3},
            {4, 5,6,7},
            {8,9}
     };
    for (int line=0; line < mult1.length;line++ ){
     for (int col=0; col < mult1[line].length;col++ ){
            System.out.print(" "+mult1[line][col]);
     }
    System.out.println();
    }
     mult1[1][4]=0; //ArrayIndexOutOfBoundsException
}
```

Конструкції управління в Java

А) простий оператор

[<вираз>];

Б) складений оператор

{<оператор1>

<оператор2> // і так далі

}

В) Умовний оператор

if (<булівський_вираз>) <оператор1> [**else** < оператор2>]

Г) цикл типу while

while (<булівський_вираз>)

<оператор>

Д) цикл типу for

for(<вираз1>; <булівський_вираз >; <вираз 2>)

<оператор>

Важливо: в Java відсутня операція , (кома), але початкова ініціалізація та вживання коми у <вираз 2> допустимо.

Конструкції управління в Java

```
for (<ім'я типу> <лок_змінна_циклу>: <змінна>)
          <оператор>
```

Е) цикл типу do_while

```
do
     <оператор>  while (<булівський_вираз>);
```

Є) розподільний оператор

```
switch (<вираз>) {
   case <конст_вираз_1>:
   [<оператори_1>]
   [break;]
   case <конст_вираз_2>:
   [<оператори_1>]
   [break;] //  і так далі
  [default: <оператори_N>
}
```

Важливо: В Java 7 в якості виразів, константних виразів допустимо (за умови, що всі вони однотипні):

- Цілочисловий тип;
- Перерахований тип;
- Рядковий (String) тип.

Конструкції управління в Java

Є) оператор break

break [<помітка>] ;

// break; // стандартна С++ інтерпретація

// break <помітка> ; // С++ інтерпретація: goto <помітка> ;

Ж) оператор continue

continue [<помітка>] ;

// continue; // стандартна С++ інтерпретація

// continue <помітка> ; // передача управління на заголовок циклу,

// що помічено поміткою.

Важливо: використання continue з поміткою дає можливість вибрати рівень вкладеності циклу, з якого буде продовжуватися виконання програми.

З) оператор return

return [<вираз>] ;

Важливо: якщо return використовується в операторі try з конструкцією finally, то спочатку виконується ця конструкція, а потім виконується перехід.

Конструкції управління в Java

Ж) **оператор try:**

```
try {
  <оператори>
  }
catch (<клас_виключення>  <аргумент>) {
   <оператори_1>
  }
catch (<клас_виключення>  <аргумент>) {
   <оператори_2>
  }
catch (<клас_виключення>  <аргумент>) {
   <оператори_3>
  }
[ finally {
   <оператори_N>
} ]
```

Важливо: return в try /catch з конструкцією finally передає управління на finally. Якщо у finally використовується return <вираз>, то попереднє значення return у цій конструкції буде замінено на нове значення.

Конструкції управління в Java

3) монопольне управління

В мові програмування JAVA при розробці потоків (thread) у деяких методах виникає необхідність організувати монопольну обробку даних (критичний сегмент). У цьому випадку програміст може скористатися префіксом synchronized:

```
synchronized (<вираз_для змінної>)  <оператор>
```

Вираз для змінної визначає об'єкт, який необхідно захистити, а оператор – це блок коду, який виконується у випадку повного (монопольного) контролю над об'єктом.

Класи

В мові Java клас існує як самостійний об’єкт, з ним можна працювати безпосередньо, або створювати екземпляри цього об’єкта.

З точки зору синтаксису клас має два компонента:

- опис класу (class declaration);
- тіло класу.

Опис класу визначається так:

[<модифікатор>] class <ім’я_класу> [extends <ім’я_суперкласу>]
[implements <список_інтерфейсів>]

Класи

Модифікатор класу

Модифікатор класу (по замовчуванню – friendly) визначає способи подальшого використання цього класу при створенні нових класів на базі цього класу та обробці виключних ситуацій. Щодо класів Java допустимі наступні модифікатори:

public – класи public доступні для всіх об’єктів, вони можуть бути розширені або використані любим об’єктом, без врахування належності до пакета. Відмітимо, що public-клас повинен зберігатися в файлах з іменем <ім’я_класу>.java .

friendly – клас з таким модифікатором може розширятися та використовуватися іншими класами, але він доступний для об’єктів, що знаходяться в тому ж пакеті.

final – клас з таким модифікатором не може мати підкласів, тобто на його основі не можна створювати нові підкласи та перевизначити методи цього класу. Цей модифікатор використовується для класів, що реалізують методи як стандарти, наприклад, клас для обслуговування мереженого протоколу.

abstract – клас з таким модифікатором має один або декілька методів, які повністю не визначені. Наприклад, маємо клас, що перевіряє орфографію певної мови. Щоб використати цей клас необхідно самостійно реалізувати метод перевірки орфографії з урахуванням специфіки мови (українська, англійська, російська).

- Відмітимо, що для абстрактних класів неможливо створювати об’єкти.

Класи: суперкласи

Суперклас для класу

На відміну від мови програмування С++, у мові JAVA при опису класу допускається не більше одного суперкласу (базового класу). Парадигма множинного успадкування в мові Java не використовується у явному вигляді. Розширяючи базовий клас ми створюємо новий клас використовуючи копію базового класу та розвиваючи його. Якщо залишити модифікатор нового класу такий як і у базового класу, то ми залишимо можливість розширення нашого класу в подальшому. Якщо у базовий клас був віртуальним класом і ми хочемо в подальшому створювати об'єкти нашого класу, то необхідно реалізувати всі методи базового класу, інакше він залишиться абстрактним. Ми також можемо залишити віртуальними методи нашого класу навіть коли базовий клас повністю реалізований. У цьому випадку ми отримаємо також віртуальний клас.

Відмітимо ще один раз, що в мові програмування Java кожний клас розглядається як об'єкт. По замовчуванню кожний клас породжується від класу **java.lang.object.** Створений нами клас *успадковує* властивості базового класу, тобто до екземпляра нового класу можна примінити методи базового класу, але не навпаки. Разом з тим кожний об'єкт нашого класу є також і об'єктом базового класу, така властивість називається *поліморфізмом*

Класи: тіло класу

Тіло класу охоплюється дужками { }.

В тілі класу необхідно визначити:

- поля класу (змінні класу),
- методи класу
- конструктори.

Визначення змінної класу передбачає, що для кожного поля необхідно вказати:

- модифікатор поля;
- тип поля;
- ідентифікатор поля.

Класи: поля класу

В мові Java для полів класу допустимі наступні модифікатори (по замовчуванню friendly) :

friendly – цей модифікатор означає, що поле доступне для всіх класів даного пакету. Але недоступне для підкласів даного класу або класів іншого пакету;

public – цей модифікатор робить поле видимим для всіх класів, підкласів та всіх пакетів. Звичайно такі поля не можуть бути в повній мірі контрольованими, їх використання в класах необхідно максимально обмежити;

protected – до полів з таким модифікатором можуть безпосередньо звертатися методи підкласів даного класу, але вони недоступні для методів поточного пакету;

private – модифікатор для найбільш захищених полів класу. Ці поля навіть недоступні для методів підкласу даного класу та поточного пакетую;

private protected – поля з таким модифікатором доступні для методів класу та для методів підкласів поточного класу;

static – цей модифікатор використовується для створення статичних полів, які зберігають значення для всіх об’єктів даного класу. Зміна static – поля в одному об’єкті приводить до зміни значення цього поля у всіх об’єктах (static – поле існує лише в одному екземплярі). Статичні поля можуть змінювати як статичні, так і нестатичні методи;

Класи: поля класу

final – поля з таким модифікатором не можна змінювати у об'єкті під час виконання програми. Ці поля можна розглядати як константні поля, а тому в JAVA допускається їх початкова ініціалізація, наприклад:

```
final int  MAX_COUNT=25;
```

Така нотація нагадує константні змінні в мові C++ (навіть і рекомендують їх записувати великими літерами). Звичайно, мати в програмі низку об' єктів, у яких використовуються final – поля, нераціонально. Рекомендують модифікатор final вживати разом з модифікатором static, наприклад,

```
static final int  MAX_COUNT=25;
```

threadsafe - модифікатор, який можна вживати перед перерахованими вище модифікаторами. Модифікатор threadsafe – це альтернатива для **synchronized** – методів, які використовуються для програмування потоків. Звичайно, в JAVA можна обійтися лише synchronized – методами, але коли потрібно захистити одне поле в об' єкті, то достатньо його специфікувати модифікатором threadsafe.

Класи: конструктори

Конструктори – це специфічні методи (інколи їх взагалі не відносять до методів), які в мові Java мають специфічні властивості, а саме:

- ім’я конструктора співпадає з іменем класу;
- конструктори не повертають значення;
- на відміну від методів класу конструктори викликаються іншим способом – в момент створення екземпляра класу, наприклад:

String myStr1 = new String(“Початкове значення змінної”);

- конструктор не можна викликати примусово до об’єкта, як метод;
- конструктор не можна визначати з модифікаторами : native, abstract, static, final, synchronize (залишаються модифікатори public, protected, friendly, private, public protected);

По аналогії з мовою С++, допускається *перевантаження конструктора (overload),* тобто створення декількох різних конструкторів, заголовки котрих відрізняються кількістю параметрів та їх типами.

Класи: конструктори

Важливо:

- на відміну від C++, програміст повинен самостійно викликати конструктор для базового класу (бажано на початку коду конструктора похідного класу, а саме:

  super(<параметри конструктора>);

- якщо в класі немає жодного конструктора, то система генерує конструктор по замовчуванню, який встановлює в 0 (null) вісі поля екземпляру класу.

Методи класу

1. Метод класу має повні права доступу до полів екземпляру класу;
2. Метод класу також маж повні права на змінні методу (в контексті прав доступу). Таким чином змінні методу не мають специфікатора доступу.
2. В мові Java в межах методу кожна змінна методу (ідентифікатор) повинна бути унікальною, тобто наступний код є помилковим:

```
int i=0;
for (int count=0; count<100; count++) {
  int i=25;
}
```

3. У випадку, коли ім’я поля співпадає з іменем параметру чи іменем змінної методу, використовуємо this.< ім’я поля >.
4. У випадку, коли ми хочемо викликати з *перевизначеного* методу (method overriding) метод базового класу, то

   super.<ім’я методу> (<список_факт_параметрів>);

   інакше – це буде рекурсія методу.

Перевизначення та перевантаження методу класу

- При розробці похідних класів мова Java дає можливість *перевизначити* методи базового класу (method overriding). При цьому необхідно запрограмувати метод з іменем, що співпадає з іменем метода базового класу, та параметрами, що співпадають по містах (по кількості та типу).

- Якщо список параметрів відрізняється від параметрів метода базового класу, то у цьому випадку мова йде про *перевантаження* (method overloading) методу з базового класу.

При *перевизначенні* методу ми не можемо зробити його більш або менш захищеним, модифікатор нового методу повинен співпадати з модифікатором методу базового класу.

При *перевантаженні* методу ми не можемо зробити його більш захищеним, ніж відповідний метод базового класу, наприклад, якщо метод базового класу має модифікатор доступу public ми не можемо для перевантаженого методу надати модифікатор private

Інтерфейси в мові Java

Інтерфейси в мові Java – це варіант можливості множинного успадкування в мові програмування С++. Інтерфейси використовуються тоді, коли необхідно описати деяку множину (набір) функціональних можливостей, що будуть використовуватися у різних класах. При цьому не фіксується спосіб реалізації цих можливостей, передбачається, що в подальшому при конкретній реалізації класів інтерфейси будуть реалізовані. Використання інтерфейсів у такій інтерпретації корисно у випадку побудови якихось загальнозначущих компонент програм.

Інтерфейс – це набір *абстрактних методів* та констант, які будуть використані іншими об'єктами. Для методів це значить, що при визначенні інтерфейсу тіло метода не буде визначено. При визначенні класу з використанням інтерфейсу необхідно обов'язково перевизначити (overriding) методи інтерфейсу. Наприклад, якщо клас реалізує інтерфейс java.lang.Runnable, він обов'язково повинен перевизначити метод run().

Інтерфейси в мові Java

З точки зору синтаксиста визначення інтерфейсів (interface declaratin) та класів досить схожі. Але тіло інтерфейса повинно мати лише поля константи та лише визначення методів (методи не повинні мати тіл). В загальному випадку визначення інтерфейса має такий синтаксис:

[public] interface <Ім’я_Інтерфейса>

[extents <Список_Інтерфейсів>]

<Список_Інтерфейсів> - це список імен інтерфейсів, записаний через кому.

По замовчуванню інтерфейси можуть реалізуватися усіма класами, що входять до конкретного пакета. Якщо інтерфейс має модифікатор public, то він повинен знаходитися у файлі <Ім’я_Інтерфейса>.java. Для зручності в роботі системи Java рекомендують інтерфейси визначати у файлах типу <Ім’я_Інтерфейса>.java.

Важливо: В концепції Java не допускається розширення інтерфейсів за рахунок класів.

Інтерфейси в мові Java

Синтаксис визначення методів в інтерфейсах наступний:

[public] <Тип_результату> <Ім'я_Методу> ([<Список_параметрів>])
[throws <Список_Виключень>];

Важливо:

- в інтерфейсах заборонено використовувати модифікатори методів крім модифікатора public, на відміну від методів, у яких по замовчуванню використовується модифікатор friendly.
- поля в інтерфейсах, не залежно як вони визначаються, автоматично отримують специфікатори public, final, static. Взагалі ці атрибути можна і не вживати при програмуванні, але склалась певна традиція і їх використовують, ще раз підкреслюючи, що це поля інтерфейсу.
- з усіх модифікаторів, що приміняються до методів, в інтерфейсах можливо використовувати лише abstract та native.
- при проектуванні методів інтерфейсу ми повинні вказати максимально потрібний нам набір виключень. Інакше при реалізації методу ми не зможемо розширити список виключень.

Інтерфейси в мові Java

Коли ми хочемо скористатися інтерфейсом при розробці деякого класу, необхідно перевизначити всі методи інтерфейсу, інакше наш клас буде абстрактним (звичайно native – методи не потрібно реалізувати).

Все вище викладене відносно інтерфейсів буде не повним, якщо не скористатися додатковими можливостями інтерфейсів як типів.

Змінну інтерфейс можна визначити як іншу змінну в програмі. Важливу роль мають змінні-інтерфейси як параметри інших функцій.

Класичним прикладом використання інтерфейсів-параметрів буде розробка метода обчислення визначеного інтеграла на проміжку від A до B функції F. Оскільки F виступає як параметр методу, ми передамо фактичну функцію через реалізацію інтерфейсу My_function.

Інтерфейси в мові Java

```
inport java.lang.Math;
public interfsce My_function {
      final double EPS=.1e-6;
      double f (double x);
      double  abs(double d);
}

public class Integral  implements My_function {
      double  abs(double d)
             { if (d < 0) return (–d); else return (d); }
      double integral (double a, double b, My_function fun) {
            double sum=0.;
            do { sum1=sum; sum+= fun.f(a);
            /*  продовжуємо розрахунок */
                        while ( fun.abs(sum-sum1) > EPS );
      }
}
```

Інтерфейси в мові Java

- Оскільки ми не перевизначили метод f() в класі Integral наш клас залишається не повністю визначеним, але в подальшому ми можемо скористатися цим класом, конкретно визначити метод f(), а метод обчислення інтегралу буде нами запрограмований.
- Важливим моментом в реалізації методів, визначених в інтерфейсах, є список виключних ситуацій та можливі варіанти його розширення.
- В Java неможливо створити екземпляр класу типу інтерфейс (метод new недопустимий. Але ми можемо створити змінну типу інтерфейс, наприклад:

```
Cloneable myListOfClones;
Manager  x=new Manager ();
myListOfClones=x;
// за умови, що клас Manager реалізує інтерфейс Cloneable
```

Важливо: операція instanceof перевіряє, чи належить об'єкт заданому класу. Разом з тим, цю операцію можна приміняти для перевірки, реалізує обёєкт заданий інтерфейс, наприклад:

```
if (x instanceof Employee) { /* перевірка     */}
if (x instanceof myListOfClones) {myListOfClones=x; }
```

Інтерфейси та абстрактні класи

Яка причина, що в Java існують і інтерфейси, і абстрактні класи. Поглянемо на приклад:

```
abstract class Comparable {
    public abstract int compareTo( Object ob) ;
}
```

Далі, клас Employee міг би розширяти абстрактний клас та реалізувати метод compareTo:

```
class Employee extends Comparable {
    public int compareTo( Object ob) {/* реалізація */};
}
```

Якщо клас Employee є підкласом Person, тоді:

```
class Employee extends Person, Comparable // а це вже помилка
```

Зважаючи на те, що клас може реалізувати декілька інтерфейсів, то:

```
class Employee extends Person implements Comparable, Cloneable
  { /* поля та методи класу */ }
```

Клас Object

```
import java.lang.Object;
public class Object  {
    // конструктор
     public Object() {  /*      */ }
    // методи класу
    protected Object   protected Object clone() { /* Creates and returns a copy
    of this object. */ }
    boolean equals (Object obj) {
            if (this == obj ) return true  else return false ;
    }
protected void finalize() { /* Called by the garbage collector on an object
    when garbage collection determines that there are no more references to the
    object. */
    }
ClassClass <?>  getClass() {/*Returns the runtime class of this Object.*/ }
int hashCode () {/*Returns a hash code value for the object.*/ }
// решту методів вивчаємо через
// http://download.oracle.com/javase/7/docs/api/
}
```

Клонування об'єктів

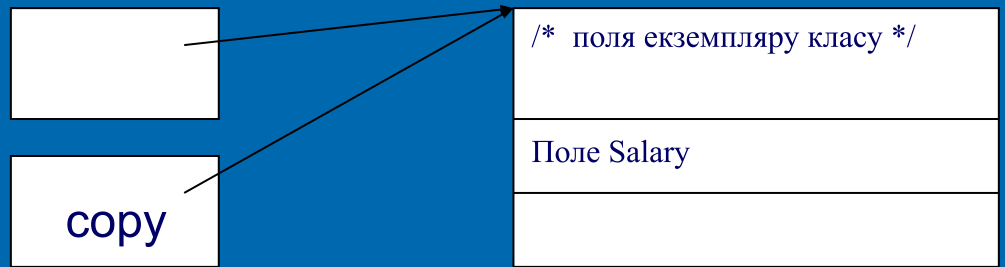
Розглянемо приклад:

Employee original = new Employee (3000 /* Salary */);

Employee copy = original ;

copy.setNewSalary(4000);

// тоді екземпляр original та copy отримають одне і теж значення Salary



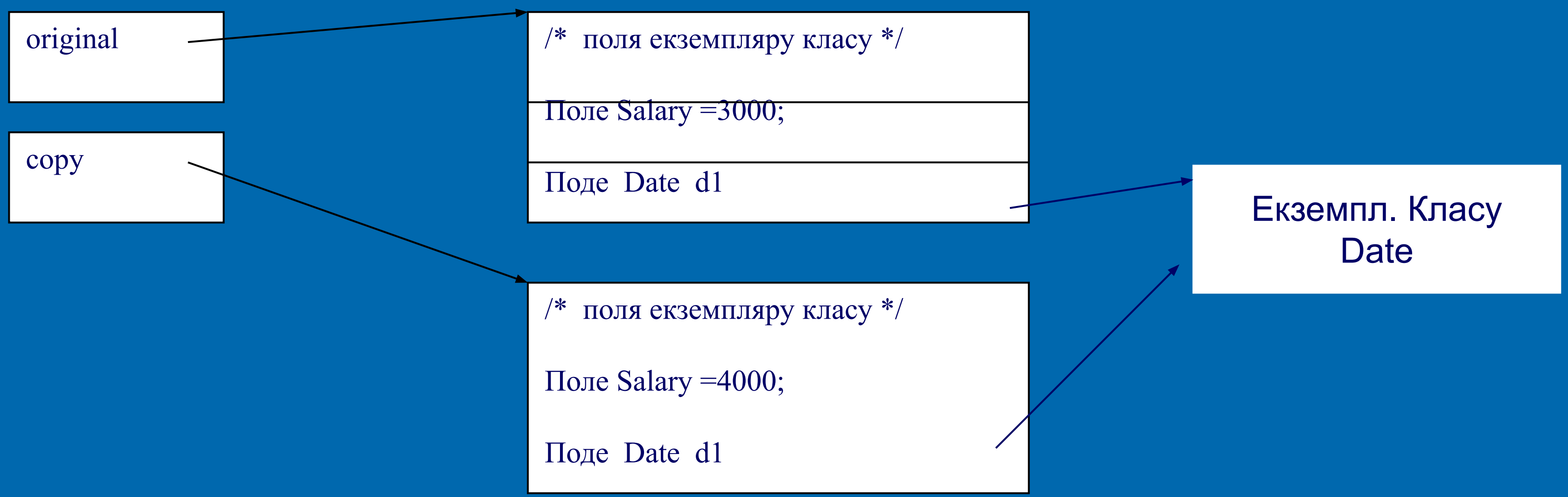
Тепер скористаємося методом clone():

Employee original = new Employee (3000 /* Salary */);

Employee copy = original.clone() ;

copy.setNewSalary(4000);

Клонування об'єктів



Після того, як скористалися методом clone(). Висновок:

- Метод clone() створює лише копіє екземпляру класу, і не більше.
- Якщо нам потрібно клонувати детально далі (“Глибинне копіювання”), ми повинні в класі Employee визначити власний метод clone(), при цьому клас повинен реалізувати інтерфейс Cloneable.

Клонування об'єктів

```
Class Employee implements Cloneable {
Public Object clone() throws CloneNotSupportedExeption {
    // створюємо екземпляр через родові класи
    Employee dataClone= (Employee ) super.clone();
    // виконуємо "глибинне клонування"
    dataClone.d1 = (Date) d1.clone();
    //  далі, по аналогії, клонуємо дані для об'єктних полів класу
    return (dataClone);
}
```

Важливо: як клонувати об'єкти повинен визначати програміст.

Розробники Java стверджують, що інтерфейс Cloneable реалізовано не більше ніж в 5% класів JDK.

Можна сказати наступне, що метод java.lang.Object.clone() через механізм рефлексії визначає екземпляр якого класу клонується, виділяє пам'ять через засоби віртуальної машини, та ініціалізує машинними нулями змінні об'єктного типу, а змінні базових типів копіює з this.<поле класу>.

Узагальнені масиви

В мові програмування C++ робота з масивами визначається як робота з масивами фіксованої довжини (довжина визначається на етапі компіляції або під час виконання програми. В Java існує відповідна аналогія:

int lenArray =20;

int [] intArray = new int [lenArray];

Разом з тим, в Java реалізовано технологію узагальнених масивів через клас ArrayList, який забезпечує динамічне управління масивом під час його обробки:

ArrayList <Employee> = new ArrayList <Employee> ();

Методи класу ArrayList:

public boolean add(E e); // Appends the specified element to the end of this list

public E remove (int index); // Removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices)

public int size(); // Returns the number of elements in this list

Узагальнені масиви

public boolean **isEmpty**(); //Returns true if this list contains no elements.

public boolean contains(Object o); //returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e))

public Object **clone**(); //Returns a shallow copy of this ArrayList instance.

public Object[] **toArray**(); // Returns an array containing all of the elements in this list

public E **get**(int index); // Returns the element at the specified position in this list.

Exception:

IndexOutOfBoundsException   if the index is out of range (index < 0 || index >= size())

public E **set**(int index, E element); //Replaces the element at the specified position in this list with the specified element. **Exception:** IndexOutOfBoundsException

public Object[] **toArray**(); // Returns an array containing all of the elements in this list in proper sequence

Детально: http://download.oracle.com/javase/6/docs/api/

Об'єктні оболонки та автоупаковка

Java надає об'єктні оболонки(Object Wrapper) для усіх базових типів. Їх імена: Integer, Long, Float, Double, Short, Byte, Character, Void, Boolean. Наприклад:

```
ArrayList <Double> arrDouble = new ArrayList <Double> ();
```

Важливо: базовий (простий) тип не може бути параметром об'єктного типу (колекції). Разом з тим, Java 5 та старше реалізує можливість автоупаковки, наприклад:

```
arrDouble.add(35); //автоупаковка arrDouble.add(new Double(35));
```

Реалізується і протилежна обробка:

```
double  dd= arrDouble.get(i); // double  dd= arrDouble.get(i).doubleValue();
```

Як результат:

```
Double  dd1=3.0;
dd1 +=10.3; //авторозпаковка, операція, автоупаковка
if (dd == dd1) // спрацює як порівняння об'єктів та видасть false
потрібно при порівнянні
if (dd.equals( dd1)) // порівняння значень об'єктів.
```

Ррозробники Java вирішили перенести перетворення текстової інформації в базові типи даних через об'єктні базові типи, наприклад:

```
String ss=‘”2011”;
int i = Integer. parseInt(ss);
```

Методи зі змінним числом параметрів

По аналогії з C++ в мові Java допустимі методи зі змінною кількістю параметрів. Синтаксично це оформляється так:

```
public class PrintStream
  { public PrintStream  printf(String  fmt, Object  … args) { /*   */ }
}
```

Компілятор перетворює список параметрів у масив параметрів відповідного типу, тобто Object []. Приклад методу зі змінною кількістю параметрів:

```
public static double max (double … value)
 { double largest =Double.MIN_VALUE;
    for (double dd: value )  if (dd > largest ) largest = dd;
     return largest ;
}
```

Коли ми викликаємо метод max(2., 41.5, 55) то компілятор передасть нашому методу наступний масив: new double [] {(2., 41.5, 55} .

Рефлексія. Клас Class.

Рефлексія – це технологія (інструментарій) динамічної роботи з Java-кодом. Рефлексія використовується для вирішення наступного переліку задач:

- Аналіз можливостей класу під час виконання програми;
- Перевірка об'єктів під час виконання програми. Наприклад, якщо об'єкт не реалізує інтерфейс Cloneable, то отримаємо відповідне виключення.
- Реалізація узагальненого коду при роботі з масивами;
- Використання об'єктів Method, які працюють подібно вказівникам в С++.

Під час виконання програми Java-машина виконує перевірку типів даних. Отримати інформацію про зареєстрований машиною клас можна за допомогою спеціального класу – Class.

```
Employee  ee;
Class c1 = ee.getClass(); // метод класу Object.
```

Рефлексія. Клас Class.

Клас *Class* не має конструктора. Замість цього клас Object конструюється автоматично Java Virtual Machine при використанні методу defineClass в класі Loader.

Деякі з методів (решта досить складні та потрібні для системних речей):

static Class<?> **forName**(String className) ; // Returns the Class object associated with the class or interface with the given string name.

ClassClass<?>[] getDeclaredClasses(); // Returns an array of Class objects reflecting all the classes and interfaces declared as members of the class represented by this Class object.

String **getName**() ; // Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this Class object, as a String.

T newInstance(); //Creates a new instance T of the class represented by this Class object.

String toString(); // Converts the object to a string.

static Class<?> **forName**(String className); // Returns the Class object associated with the class or interface with the given string name.

Рефлексія. Клас Class.

Продемонструємо «чудо» - властивості рефлексії:

String ss="java.utils.Date";

Date dd = (Date) Class.forName(ss).newInstance();

Для використання інструменту рефлексії детально потрібно вивчити класи:

- Java.lang.Class, Java.lang.reflect.Constructor, Java.lang.reflect.Field, Java.lang.reflect.Method,

Рефлексія та метод equals()

```
Class Employee extends Person
{
    Employee () {/* конструктор */}
    public boolean equals(Object ob)
  {  if (this == ob)  return true; // швидка перевірка на рівність об'єктів
    if (ob == null) return false; // якщо параметр методу null
    if (getClass() != ob getClass() ) return false; // «жорстко»: по імені
    // if (! (ob  instanceof Employee) ) return false; «м'яко»: по типу
    Employee  tmp = (Employee ) ob;
    return salary== ob.salary && this.super(ob);
  }
double salary;
}
```

Внутрішні класи

В внутрішні класи (inner class) – це клас, що визначений всередині іншого класу. Внутрішній клас має наступні три властивості:

- Об'єкт внутрішнього класу має повний доступ до даних об'єкта, в якому він визначений;
- Внутрішній клас можна будувати від інших класів цього пакета;
- Ніхто інший, окрім класу, в якому визначений внутрішній клас, не має інформації про його існування.
- Відношення “внутрішній” - це відношення між класами, а не між об'єктами.Коли ми створюємо обёєкт внутрішнього класу, компілятор додатково створює в екземплярі внутрішнього класу поле, а код конструктора добавляє команду, що пов'язує екземпляр внутрішнього класу з екземпляром зовнішнього класу. Це поле має назву <ім'я зовнішнього класу>.this. Таким чином, доступ до полів зовнішнього класу виконується як:

<ім'я зовнішнього класу>.this.<ім'я поля>

Внутрішні класи

По аналогії, можна більш конкретно записати виклик конструктора внутрішнього класу:

this.new <ім'я внутрішнього класу> ([<параметри>]);

Загальна філософія питання внутрішнього класу:

- В JVM не працює з поняттям "внутрішній клас", це прерогатива компілятора. Компілятор на синтаксичному рівні перетворює внутрішній клас у зовнішній клас, використовуючи спецсимвол $, зокрема, ім'я внутрішнього класу інтерпретується синтаксично:

<ім'я зовнішнього класу>$<ім'я внутрішнього класу>

Виникає питання, якщо компілятор перетворює внутрішній клас у зовнішній клас, то яким чином наш новий зовнішній клас має доступ до усіх полів класу "сусіда": компілятор створює у внутрішньому класі методи, що надають значення полів для внутрішнього класу.

Локально внутрішні класи

Локально внутрішній клас – це клас, який визначається локально у окремому методі нашого класу. Оскільки такий клас є локальним у методі, то модифікатор класу не вживається, наприклад:

```
public void start ( )
{  class TimePrinter implements ActionListner
     {


     }
}  // кінець методу
```

Важливо: локальні класи повністю приховані від зовнішнього світу, навіть від від того класу, у методі якого вони визначені.

Локальні класи мають доступ не тільки до полів зовнішнього класу, але й до змінних методу, у якому вони визначені. Але такі локальні змінні (в тому числі і параметри методу) повинні бути визначені як final.

Анонімні внутрішні класи

Розглянемо приклад локально внутрішнього класу:

```
….
int counter=0;
Data datas[] = new Data (100);
for ( int i=0; i< 100; i++)
  datas[i]= new Data ()
    {  public int compareTo (Data other)
         {counter++;    // помилка: змінна без специфікатора final
       return super.compareTo(other);
         }
     };
Arrats.sort(datas);
System.out.println((“Порівнянь ”+ counter);
```

Вирішення проблеми: через масив з одним елементом:

```
final int  [] counter= new int [1];   // замість int counter=0;
counter[0]++;    //  замість counter++; посилку не міняємо, але значення
              // можна змінювати.
```

Анонімні внутрішні класи

Оскільки анонімний клас не має імені, конструктор такого класу відсутній, конструювання класу (його параметри) можна задати через конструктор супер_класу:

```
new  СупеТип (<параметри конструювання>)
{
// методи внутрішнього класу та дані
};
```

Висновок: анонімні внутрішні класи за рахунок скорочення тексту програми на декілька рядків роблять її незрозумілою. Це одна з альтернатив в Java від якої можна завжди відмовитися, але для читання чужих текстів цю технологію потрібно знати.

Виключення в Java

В програмуванні ми зустрічаємо наступні ситуації:

- Метод (підпрограма) дізнається про помилку, але не знає як не неї реагувати;
- Метод передбачає, що у програмі можуть бути помилки під час виконання, але не знає які;
- Програміст хоче розробити надійний програмний комплекс, що забезпечить збереження даних завжди при виконанні програми.

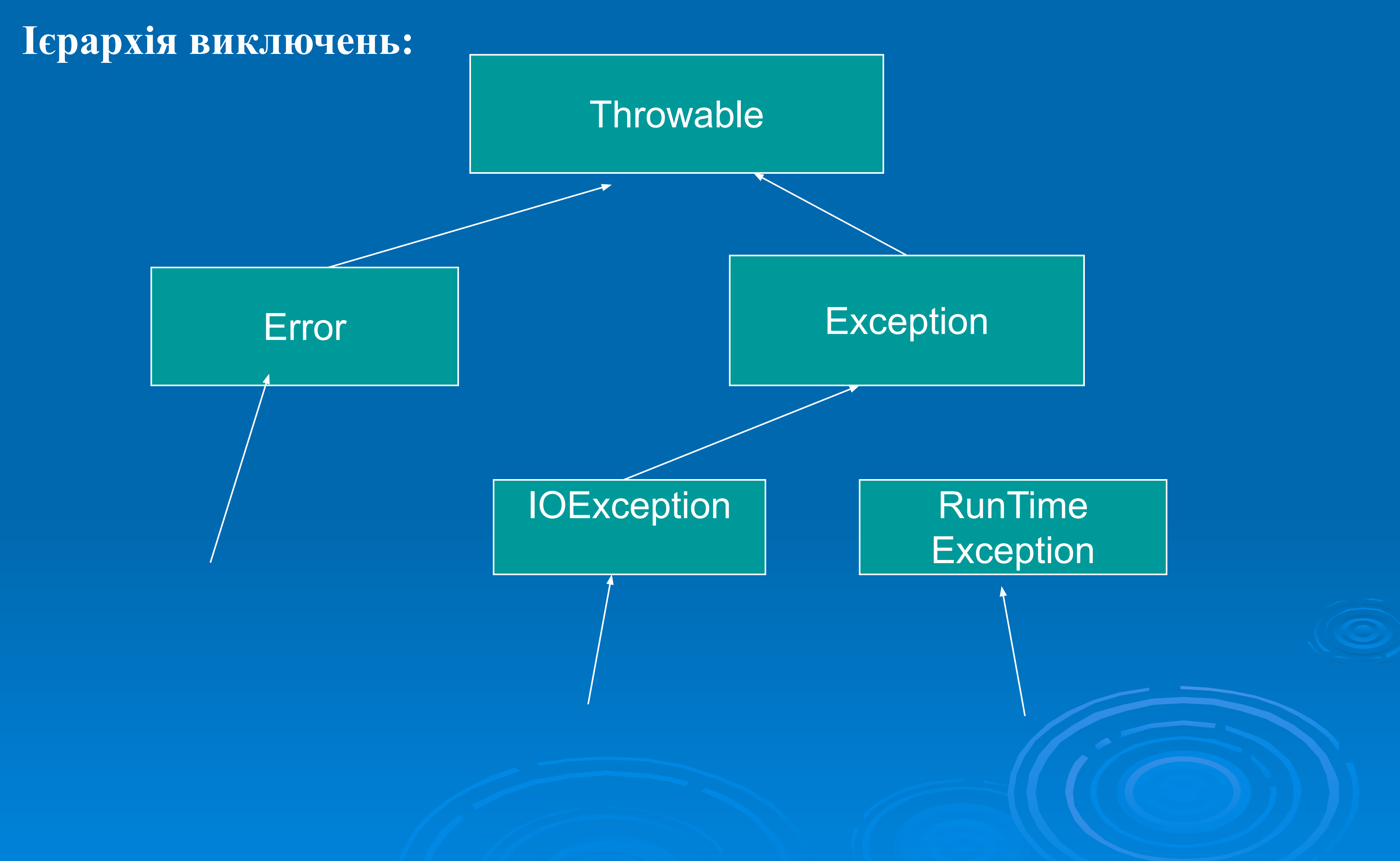
Які помилки бувають:

- Помилки вводу/виводу.
- Помилки у роботі обладнання та фізичні обмеження;
- Помилки програмування.

Технологія роботи з помилками під час виконання програми співпадає з відповідними технологіями в мові C++. Але є своя специфіка.

Виключення в Java

Ієрархія виключень:



Клас Throwable

В клас Throwable є три корисних методи:

getMassage() - Повертає рядок з детальною інформацією про виключення;

toString() - Перетворює об'єкт в рядок, який потів можна вивести на екран термінала;

printStackTrace() - Відображає ієрархію викликів методів, що привели до виключення.

В класі Exception відсутні будь-які корисні методи, окрім конструкторів. Від класу Exception ієрархія розгалужується на три основних групи:

- набір класів-виключень, безпосередньо породжених від Exception;
- набір класів-виключень періоду виконання (runtime);
- набір класів-виключень вводу/виводу.

Виключення в Java

Ієрархія Error – це помилки, які виникають у випадку недостатньої кількості ресурсів. Жоден об'єкт такого типу прикладна програма згенерувати не може. Дії прикладної програми теж обмежені, вона може лише повідомити про помилку та завершити роботу. Таким чином, програмісти повинні зосередитися на виключеннях по ієрархії Exception. Вони діляться виключення, які похідні від RuntimeException та решта. RuntimeException – видаються у випадку: невірного перетворення типів, виходу за межі масиву, спробі звернутися до об'єкту по посилці null – вони виникають з вини програміста. Виключення Error та RuntimeError називаються неконтрольованими, решта – контрольовані. Для контрольованих виключень компілятор завжди перевіряє наявність обробника.

Контрольовані виключення

У заголовку методу вказуються всі контрольовані виключення, які він генерує:

[<специфікатор_доступу>] [<модифікатор>] [<тип_результату>]
<ім’я_методу> ([<список_параметрів>])
[throws <список_виключень>]

Згенерувати виключення можна лише методом
throw <екземпляр класу виключення>;

Наприклад:

```
EOFException  e= new EOFException ();
throw e;
```

Або

```
throw new EOFException ();
```

Перехоплення виключення

```
try {
    <оператори>
  }
  catch (<клас_виключення>  <аргумент>) {
    <оператори_1>
  }
  catch (<клас_виключення>  <аргумент>) {
    <оператори_2>
  }
  catch (<клас_виключення>  <аргумент>) {
    <оператори_3>
  }
  [ finally {
    <оператори_N>
  } ]
```

Важливо: В списку конструкцій catch порядок виключень принциповий, оскільки тут іде “м'яка” перевірка

```
<аргумент> instanceof <клас_виключення>
```

Створення власних класів-виключень

Для створення власного класу виключення необхідно:
створити клас для даного виключення як розширення базового класу Exception;
в деякому класі запрограмувати метод, який буде при виникненні помилки при обробці даних генерувати виключення з параметром – об'єктом нашого класу-виключення. Наприклад:

```
public class NumberRangeException  extends Exception
 {
      public NumberRangeException (String srt) {
        super (str);       }
      }

public class MyTestException
  {    final static int MAX=10000;
       private int data=0;
      public MyTestException( int dat ) throw NumberRangeException
      { // analyse data
        if ( dat < 0 ||  dat > MAX)
               throw (new NumberRangeException ("Err21"));
         }
      }
```

Створення власних класів-виключень

Відмітимо, що коли ми наперед не знаємо яке виключення ми отримаємо в нашій програмі, то можна завжди скористатися конструкцією

```
catch (Exception e)  {  /*     */  }
```

потім спробувати динамічно визначити тип об'єкта, який ми отримуємо.

Звичайно, при розробці програмного забезпечення в Java системах можуть виникнути ряд помилок інтеграції середовища. Java в момент взаємодії його з іншими середовищами та так звані внутрішні помилки. Для цього в JAVA додатково введено клас Error, який є похідним від Throwable

. Звичайно про цей клас виключень теж потрібно знати (перехопити такі виключення неможливо, але їх наявність під час виконання Java програм свідчить про фатальність у ситуації, що виникла).

Основні типи виключень

ArithmiticException - математичні помилки, наприклад: ділення на нуль

ArrayIndexOutOfBoundException - невірний індекс масиву

ArrayStoreException - спроба записати в масив невірний тип даних

FileNotFoundException - звернення до неіснуючого файлу

IOExeption - помилки вводу-виводу

NullPointerException - посилання на null – об’єкт

NumberFormatException - невірне перетворення рядка в число

OunOfMemoryException - недостатньо пам’яті при розміщення нового об’єкта

SecurityException - спроба аплета виконати дію, що заборонена в браузері

StackOwerflowException - переповнення стека системи Java

StringIndexOutOfBoundException - спроба звернутися до неіснуючого індексу в рядку

Ієрархія виключень

Ім'я виключення	**Пакет**	**Ім'я виключення**	**Пакет**
Exeption	java.lang	- **RuntimeException**	java.lang
- AWTException	java.awt	-- ArithmeticException	java.lang
- NoSuchMethodException	java.lang	-- ArrayStoreException	java.lang
- InterruptedException	java.lang	-- ClassCastException j	ava.lang
- InstallationException	java.lang	-- IllegalArgumentException	java.lang
- ClassNotFounfException	java.lang	-- IllegalThreadStateIllegalException	java.lang
- CloneNotSupportException	java.lang	-- NumberFormatException	java.lang
- IllegalAccessException	java.lang	-- IllegalMonitorStateException	java.lang
- **IOException**	java.io	-- IndexOutOfBoundsException	java.lang
-- EOFException	java.io	--- ArrayIndexOutOfBoundsException	java.lang
-- FileNotFoundException	java.io	--- StringIndexOutOfBoundsException	java.lang
-- InterruptedIOException	java.io	-- NegativeArraySizeException	java.lang
-- UTFDataFormatException	java.io	-- NullPointerException	java.lang
-- MalformadURLException	java.net	-- SecurityException	java.lang
-- ProtocolException	java.net	-- EmptyStackException	java.util
-- SocketException	java.net	-- NoSuchElementException	java.util
-- UnknownHostException	java.net		
-- UnknownserviceException	java.net		

Узагальнення (Generics)

Узагальнене програмування – це створення коду, який багато-разово можна використати для роботи з об'єктами різного плану. До появи Java 5.0 узагальнене програмування реалізовувалось через інструмент успадкування. Всі екземпляри класів мали одного пращура – Object. Таким чином, клас ArrayList – це просто масив посилок на Object.

```
public class ArrayList // до Java 5.0
{public  Object get (int i) { /*    */ }
 public ArrayList () { /* конструктор */}
public ArrayList (int  items) { /* конструктор */}
private Object [ ] elementData; // це просто масив посилок на Object
}
```

В подальшому, якщо ми організовуємо роботу з ArrayList, то потенційно його елементами можуть бути екземпляри різних класів. У цьому випадку ми повинні бути впевненими, що наш масив містить об'єкти (екземпляри) відповідного класу. Тоді ми звертаємося до рефлексії та пробуємо перевірити:

Узагальнення (Generics)

```
try {
int index =10;
Object <змінна> = Object get (index ) ;
if  (!  ( < змінна >  instanceof <конкретний клас>)) return false;
// це наш об'єктик
<конкретний клас>  tmp = (<конкретний клас>) <змінна>;
// подальша обробка даних
}
catch (ArrayIndexOutOfBoundsException  ee )
{ /* обробка  */ }
```

Звичайно, у конкретному випадку ми хочемо працювати з ArrayList, що містить лише елементи String та не задумуватися про якісь перетворення. Звичайно, за потреби ми також можемо працювати з елементами даних різних класів, тобто попередній випадок також потрібно мати на увазі.

Узагальнення (Generics)

Узагальнений клас - це клас з однією чи більше змінних типу.

Наприклад:

```
public class Pair <T>
{ public Pair ( ) { first=null; second = null; }  // конструктор
 public Pair (  T fItem, T secItem) {first= fItem; second = secItem; }
 public T getFirst ()  { return first; }

private  T first;
 private T second;
}
```

Тепер ми можемо створити:

- Пари дійсних чисел: Pair <Double> = new Pair <Double> ();
- Пари рядків: Pair <String> = new Pair <String> (“кібернетик”, “3 курс”);

Java допускає декілька змінних типів. Тоді синтаксис наступний:

```
public class Pair <T, D>  { /* клас */  }
```

Узагальнені методи

Java дозволяє створювати узагальнені методи – це методи, що визначаються у звичайних класах, наприклад:

```
class ArrayAlg {
public static <T> T getMidle ( T [ ]  a ) {
 return a[ a.length / 2];
}
}
```

Обмеження змінних типів:

1. Змінними типу можуть бути лише об'єктні типи.
2. Інколи, ми хочемо мати на увазі типи, що характеризуються певними властивостями, наприклад:

- тип повинен бути похідним від наперед заданого типу;
- реалізувати наперед визначений інтерфейс. Тоді:

Обмеження змінних типів

public class Pair <T extends Comparable> { /* клас */ }

Семантика: тип T повинен обов'язково бути похідним від Comparable. // у загальному випадку від Object

Тут ми бачимо, Comparable – це інтерфейс. Тоді виникає питання, а чому extends?. Відповідь: такий синтаксис, але ми побачимо, що це також корисно, оскільки синтаксис обмеження наступний:

extends <список типів обмеження>

Наприклад: extends Person & Comparable & Cleanable

Важливо: в обмеженні можна використовувати довільну кількість інтерфейсів, але обмежуючий клас повинен бути один та перший.

Узагальнений код та JVM

1. Віртуальна машина не працює з узагальненими об'єктами.
2. Компілятор перетворює наші узагальнені методи чи класи на загальні, тобто там де потрібно підставляється клас Object або типом першого обмеження.

Така процедура називається “підчистка” або

створення “сірого класу”.

Висновок:

вся робота по перетворенню типів до потрібних виконується компілятором автоматично.

Масиви параметризованих типів недопустимі:

```
Pair <String> [ ] table = new Pair <String> [ 20];
// помилка, після підчистки ми матимемо Pair [ ] ,
що значить Object [ ].
```

Заборонено створювати екземпляри змінних типу:

new T () , new T [ ], або T.class.

Також заборонено вживання змінних типу в статичному контексті:

```
Public static  T get() { /*     */   }
```

Wildcard типи

Питання жорсткого контролю типів не завжди задовольняє кінцевого користувача (програміста). Тоді в Java запропоновано наступний синтаксис:

```
class  Pair <? extends  Employee> {   /*     */   }
```

Тобто для цього класу параметром може виступати любий параметр типу, що є підкласом Employee, такий як:

Pair <Manager> але не Pair <String>.

Додатково Java допускає наступний синтаксис:

```
class  Pair <? super Employee> {   /*     */   }
```

Що дає отримати обмеження по супертипу.

Колекції

В мові Java 1 передбачався невеликий набір класів для роботи з колекціями: Vector, Stack, Hashtable, Bitset та інтерфейс Enumeration.

При цьому вище перераховані класи орієнтувалися на збереження Object. Це було чудове, елегантне рішення, яку не вимагали додаткових затрат та ресурсів. Разом з тим, ці класи мали один суттєвій недолік: програмісту доводилося кожного разу динамічно перевіряти реальний тип об'єкта, який був вийнятий (extracted) з колекції.

В новій реалізації колекцій в Java 2 розробники зрозуміли, що необхідно інтерфейси відділити від реалізації.

Це можна продемонструвати на наступному прикладі:

```
interface Queue <E>  // спрощена форма інтерфейсу
{
    void add( E item);
    E remove();
    int size();
}
```

В цьому випадку інтерфейс не дає ніякого уявлення про те, як буде побудована черга, наприклад: через узагальнений масив чи через список.

Колекції

Висновок, реально починаючи з Java 5.0 м модель колекції введено два базових поняття: Collection та Iterator.

```
public interface Collection <E>
{
    boolean add (E item);
    Iterator <E> iterator();
// тут описані і інші методи
}

public interface Iterator <E>
{
    E next();
     boolean hasNext();
    void remove ();
}
```

Приміряючи багаторазово метод hasNext() ми можемо “відвідати” кожний екземпляр колекції.

Колекції

Наприклад:

```
Collection <String>  c = new ArrayList ();
//  наповнюємо елементами даних  колекцію с
Iterator <String> iter = c.iterator();
while  (iter.hasNext() )
 {
String element = iter.next();
// обробка даних
}
```

Звичайно, починаючи з Java 5.0 ми можемо скористатися новою конструкцією управління for

```
For (String element :  c )
{
// обробка даних
}
```

Колекції

Виходячи з попереднього прикладу, бачимо:

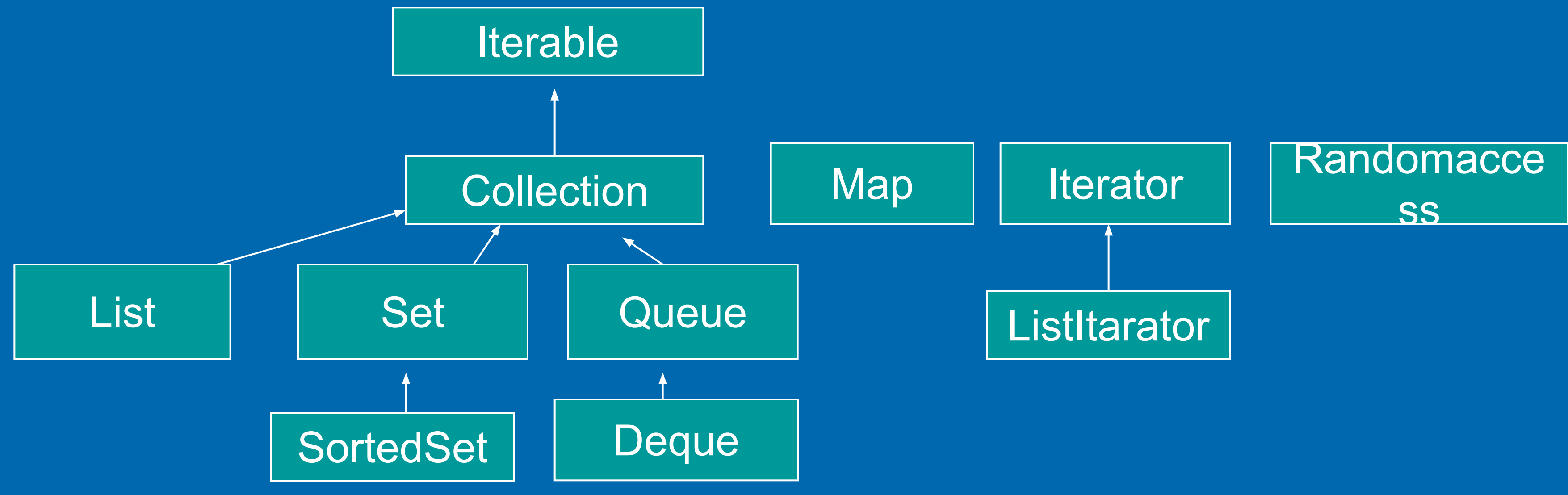
- Спочатку ми перевіряємо, чи є елементи в колекції , точніше: чиє в колекції наступний елемент
- Потім ми вибираємо наступний елемент

Таким чином, встановлюючи початкове значення ітератора, ми стаємо перед першим елементом у колекції.

Тепер зрозуміло, щоб видалити елемент в колекції, необхідно стати перед ним. Ставши перед елементом, ми отримаємо до нього доступ через метод next(), проаналізуємо дані, якщо потрібно, видалимо елемент.

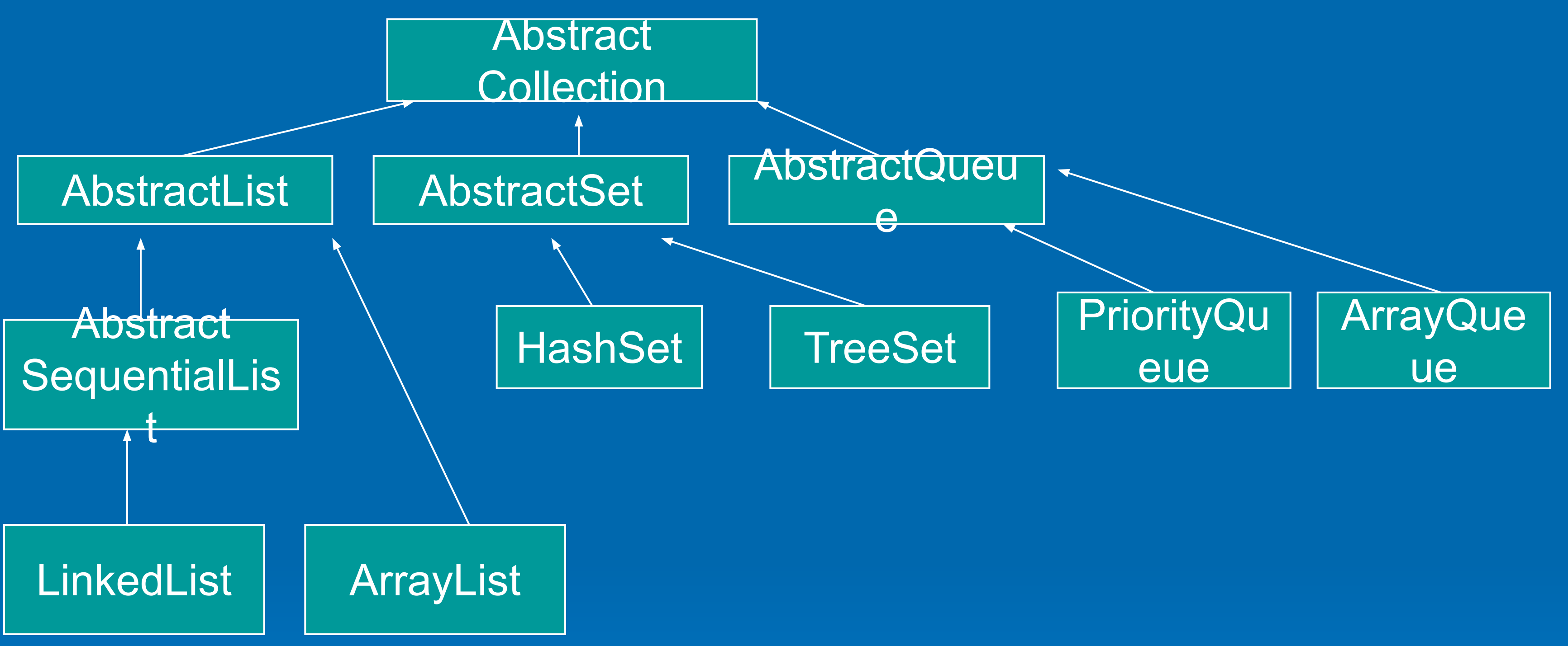
Колекції

Інтерфейси каркасу колекцій:// не все



Колекції

Класи із каркасу колекцій: // не повністю



Колекції

Колекції, що в даний час реалізовані в мові Java:

- ArrayList – узагальнений масив. Послідовність, що динамічно розтягується та стискається.
- LinkedList – Упорядкована послідовність (список), який ефективно забезпечує операції вставки та видалення
- ArrayDeque – Упорядкована послідовність (циклічний список), який ефективно забезпечує операції вставки та видалення
- HashSet – Неупорядкована послідовність (множина) елементів (без повтору)
- TreeSet – Відсортований набір (відсортоване дерево)
- EnumSet – Множина значень перерахованого типу: enum
- LinkedHashSet – Впорядкована послідовність (множина) елементів (без повтору)
- PriorityQueue – Список, що забезпечує ефективне видалення найменшого елемента
- HashMap – Асоціативна таблиця. Зберігає асоціацію “ключ/значення”
- TreeMap – Карта з сортованими ключами
- EnuMap – Карта, у якої ключі відносяться до перерахованого типу: enum
- LinkedHashMap – Карта, що запам'ятовує порядок, у якому добавлялися елементив
- WeakHashMap – Карта, яка надає GC додаткові повноваження по видаленню даних
- IdentifyHashMap – Карта з ключами, що порівнюються ==, а не equals().

Колекції

Деякі проблеми щодо колекцій:

1. Як, яким чином TreeSet знає щодо порядку сортування даних. Для цього він користується інтерфейсом Comparable. Цей інтерфейс визначає один метод:

```
public interface Comparable <T>
{
int compareTo (T other) ;
}
```

Як відомо, цей метод за специфікацією Java надає нам три варіанти відповіді (не важливо яке там значення), a. compareTo(b):

а) значення > 0; // a > b, коли a “більше за” b

б) значення == 0; // a == b, коли a “рівне” b

б) значення < 0; // a < b, коли a “менше за” b

Звичайно, ви можете скористатися методом equals(). У цьому випадку вам теж доведеться переписами (overwrite) цей метод.

ПІДДІАПАЗОНИ

Для деяких колекцій є можливість виділяти піддіапазони:

LinkedList stuff = …..:

LinkedList grope=stuff.sublist(10,20); //в grope помістяться елементи з 10 по 19 включно, по аналогії функції substring(….) класу String;

grope.clear(); // при цьому grope2 не матиме жодного елемента та з stuff буде видалено елементи .

Звичайно, якщо будемо добавляти елементи в grope, то вони з'являться і в stuff.

НЕМОДИФІКОВАНІ КОЛЕКЦІЇ

Якщо ми хочемо з часом заблокувати нашу колекцію від спроби модифікувати, то Java надає наступні методи для блокування:

Collections.unmodifiableCollection

Collections.unmodifiableList

Collections.unmodifiableSet

Collections.unmodifiableSortedSet

Collections.unmodifiableMap

Collections.unmodifiableSortedMap

В подальшому, для методу look() ми блокуємо модифікацію:

Look(new Collections.unmodifiableList(stuff)) ;

Колекції

Старі колекції Java 1.0:

List - класичний список;

Vector – масив, у який реалізує неміряну кількість методів:

addadd, addAlladd, addAll, addElementadd, addAll, addElement, capacityadd, addAll, addElement, capacity, clearadd, addAll, addElement, capacity, clear, cloneadd, addAll, addElement, capacity, clear, clone, containsadd, addAll, addElement, capacity, clear, clone, contains, containsAll,

copyIntocopyInto, elementAtcopyInto, elementAt, elementscopyInto, elementAt, elements, ensureCapacitycopyInto, elementAt, elements, ensureCapacity, equalscopyInto, elementAt, elements, ensureCapacity, equals, firstElementcopyInto, elementAt, elements, ensureCapacity, equals, firstElement, getcopyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode,

indexOf indexOf, insertElementAt indexOf, insertElementAt, isEmpty indexOf, insertElementAt, isEmpty, iterator indexOf, insertElementAt, isEmpty, iterator, lastElement indexOf, insertElementAt, isEmpty, iterator, lastElement, lastIndexOf,

listIteratorlistIterator, removelistIterator, remove, removeAlllistIterator, remove, removeAll, removeAlElements,

removeElementremoveElement, removeElementAtremoveElement, removeElementAt, removeRangeremoveElement, removeElementAt, removeRange, retainAllremoveElement, removeElementAt, removeRange, retainAll, setremoveElement, removeElementAt, removeRange, retainAll, set, setElementAt,

setSizesetSize, sizesetSize, size, subListsetSize, size, subList, toArraysetSize,