# ПАКЕТИ JAVA

Лекція 5.2
доц. кафедри Інформатики
Сінельнікова Т.Ф.

# Зміст

- Поняття пакета
- Організація та особливості доступу до класів пакетів
- Права доступу до полів і методів класу
- Пакет java.lang
- Пакет java.util: структура колекцій
- Пакет java.util.zip
- Пакет java.util.jar
- Пакет java.io

# Поняття пакета

Розробники Java включили в мову додаткову конструкцію - пакети (packages).

Всі класи Java розподіляються по пакетах.

Крім класів пакети можуть включати в себе інтерфейси і вкладені подпакетах (subpackages).

Утворюється деревовидна структура пакетів і подпакетах.

Ця структура в точності відображається на структуру файлової системи. Всі файли з розширенням class (містять байт-коди), що утворюють пакет, зберігаються в одному каталозі файлової системи.

Подпакетах зібрані в підкаталоги цього каталога.

Кожен пакет утворює одне простір імен (namespace). Це означає, що всі імена класів, інтерфейсів і подпакетах в пакеті повинні бути унікальні. Імена в різних пакетах можуть збігатися, але це будуть різні програмні одиниці.

Таким чином, жоден клас, інтерфейс або подпакетах не може виявитися відразу в двох пакетах.

# Організація та особливості доступу до класів пакетів

Якщо треба використовувати два класи з однаковими іменами з різних пакетів, то ім'я класу уточнюється ім'ям пакету: пакет, клас. Таке уточнене ім'я називається повним ім'ям класу (fully qualified name).

Пакетами користуються ще й для того, щоб додати до вже наявних прав доступу до членів класу private, protected і public ще один, "пакетний" рівень доступу.

Якщо член класу не відмічений жодним з модифікаторів private, protected, public, то, за умовчанням, до нього здійснюється пакетний доступ (default access), а саме, до такого члену може звернутися будь-який метод будь-якого класу з того ж пакета. Пакети обмежують і доступ до класу цілком - якщо клас не помічений модифікатором public, то всі його члени, навіть відкриті, public, не буде видно з інших пакетів.

# Права доступу до полів і методів класу

| | Клас | Пакет | Пакет та подкласи | Всі класи |
|---|---|---|---|---|
| private | + | | | |
| package | + | + | | |
| protected | | + | + | * |
| public | + | + | + | + |

Особливість доступу до protected-полям і методам з чужого пакету відмічена зірочкою.

# Пакет *java.lang*

java.iang автоматично імпортується в усі програми.

Він містить класи та інтерфейси, які є фундаментальними фактично для всього Java-Програмування.

Це найбільш широко використовуваний пакет Java.

# Клас *Number*

Абстрактний клас Number визначає суперклас, який реалізується класами-оболонками числових типів byte,short,int,long,float і double.

У класі Number існують абстрактні методи, які повертають значення у формі *об'єкту* кожного числового формату.

Наприклад, doublevalue () - Повертає значення у формі об'єкта типу double,floatvalue () - Повертає значення у формі об'єкта типу float, І т. д.

# Оболонки *Double* і*Float*

Double iFloat - Це оболонки (тобто класи) для значень з плаваючою точкою типу double ifloat, Відповідно.

Конструктори Float такі:

- Float (double num)
- Float (float num)
- Float (String str) throws NumberFormatException

Можна створювати Float-Об'єкти зі значеннями типу float або double. Їх можна також конструювати з строкового подання чисел з плаваючою крапкою.

Конструктори Double наступні:

- Double (double num)
- Double (String *str)* throws NumberFormatException
- Double-Об'єкти можна створювати або з числовим значенням типу double, Або з рядковим значенням, що містить символьне представлення числа з плаваючою крапкою.

# Методи *islnfinite() I isNaN()*

В оболонках Float iDouble визначено методи isInfinite() iisNaN (), Які допомагають при управлінні двома спеціальними значеннями типу double ifloat.

Обидва методи перевіряють два унікальних значення, визначених технічними специфікаціями IEEE для числових даних з плаваючою точкою: нескінченність і NaN (Not a Number - Не число).

Метод Isinfinite () повертає true, Якщо перевіряється значення нескінченно велике або мало за величиною. Метод isNaN () повертає true, Якщо перевіряється значення - не число.

# Оболонки *Byte, Short, Integer* і*Long*

Класи Byte, Short, Integer iLong - оболонки для типів byte, short, int ilong, Відповідно. Їх конструктори такі:

- Byte (byte num)
- Byte (String *str)* throws NumberFormatException
- Short (short num)
- Short (String *str)* throws NumberFormatException
- Integer (int num)
- Integer (String *str)* throws NumberFormatException
- Long (long num)
- Long (String *str)* throws NumberFormatException

# Перетворення чисел в рядки і назад

Однією з найбільш звичайних робіт в програмуванні є перетворення строкового подання числа в його внутрішній, двійковий формат.

У класах Byte,Short,Integer iLong визначено методи parseByte (), parseShort (), parseInt() I parseLong().

Ці методи повертають byte-, short-, int- Або long- Еквівалент числовий рядки, з якою вони викликаються. (Подібні методи також існують для класів Double iFloat.)

# Оболонка *Character*

Клас Character - Проста оболонка для типу char.

Конструктор класу character має наступну форму:

- Character(char *ch)*

де *ch* - Символ, який буде "обгорнутий" в створюваний об'єкт типу Character.

Щоб отримати char-Значення, що міститься в character-Об'єкті, викличте метод charValue ().

Сигнатура цього методу:

- char charValue ()

Метод повертає char-Значення, тобто символ.

# Оболонка *Boolean*

Boolean - Дуже тонка оболонка навколо boolean-Значень, яка є корисною головним чином тоді, коли ви хочете передати змінну boolean за посиланням.

Вона містить константи TRUE iFALSE, Які позначають істинні і помилкові Boolean-Об'єкти. Клас Boolean визначає також поле TYPE, Яке є об'єктом типу Class для boolean.Boolean визначає наступні конструктори:

- Boolean (boolean *bootValue)*
- Boolean (String *bootstring)*

У першій версії параметр *bootvalue* повинен бути або true, Або false. У другій версії, якщо параметр *bootstring* містить рядок "true"(У верхньому або нижньому регістрі), то новий Boolean-Об'єкт буде true. Інакше - false.

# Клас *Void*

Клас void має єдине поле TYPE, Яке містить посилання до об'єкта типу Class для типу void.

Екземпляри цього класу не створюються.

# Клас *Process*

Абстрактний клас Process інкапсулює *процес,* т. е, що виконує програму.

Він використовується перш за все як суперклас для типу об'єктів, що створюються методами exec () в класі Runtime.

# Клас *Runtime*

Клас Runtime інкапсулює виконавчу середу Java (Середу часу виконання).

Ви не маєте можливості створювати об'єкт типу Runtime.

Однак ви можете отримати посилання на поточний Runtime-Об'єкт, викликаючи статичний метод Runtime.getRuntime ().

Отримавши посилання, ви зможете викликати деякі методи, які керують станом і поведінкою віртуальної машини Java (JVM).

Аплети та інші ненадійні коди зазвичай не можуть викликати будь-якої Runtime-Метод без того, щоб виникло виключення типу SecurityException.

# Управління пам'яттю

Хоча Java забезпечує автоматичну збірку "сміття", іноді потрібно знати, як великий обсяг об'єктної динамічної пам'яті (object heap), І скільки її залишилося вільною.

Ви можете використовувати цю інформацію, наприклад, для перевірки ефективності вашого коду або приблизної оцінки кількості створюваних об'єктів (деякого типу).

Щоб отримати ці значення, застосовуйте методи totalMemory() І freeMemory().

# Клас *System*

Клас System містить сукупність статичних методів і змінних.

Стандартний введення, висновок і висновок помилок виконавчої системи Java зберігаються в змінних in,out ierr, Відповідно.

Багато з методів цього класу викидають виключення типу securityException, Якщо операція не дозволяється менеджером безпеки (security manager).

# Використання методу *arraycopy ()*

Метод arraycopy () можна використовувати для швидкого копіювання масивів будь-якого типу з одного місця в інше.

Він працює набагато швидше, ніж еквівалентний звичайний цикл Java.

# Для будь-якого середовища Java 2 доступні наступні властивості

- file.separator
- Java.class.version
- Java.class.path
- Java.home
- java.specification.name
- java.vm.vendor
- java.specification.vendor
- Java.vm.version
- java.specification.version
- line.separator
- java.vendor
- os.arch
- java.vendor.url
- os.name
- java.version
- os.version
- java.vm.name
- path.separator
- java.vm.specification.name
- user.dir
- java.vm.specification.vendor
- user.home
- java.vm.specification.version
- user.name

# Клас *Object*

Клас Object є суперкласом всіх інших класів.

# Клас *Class*

Class інкапсулює стан об'єкта або інтерфейсу під час виконання програми (так зване *run-time-Стан).*

Об'єкти типу Class створюються автоматично під час завантаження класів.

Ви не можете явно оголошувати об'єкт типу Class.

Але вам дозволено *отримати* об'єкт типу Class, Викликаючи метод getClass () класу Object.

# Клас *ClassLoader*

Абстрактний клас ClassLoader визначає, як завантажуються класи.

Ваше додаток може створювати підкласи, що розширюють ClassLoader, Реалізуючи його методи.

Це дозволяє завантажувати ваші класи дещо іншим способом, ніж при звичайній їх завантаженні виконавчою системою Java.

# Клас *Math*

Клас Math містить всі функції з плаваючою крапкою, які застосовуються в геометрії і тригонометрії, а також кілька універсальних методів.

В Math визначені дві константи типу double:

- E(Приблизно 2.72) і PI (Приблизно 3.14)

# Інтерфейс *Runnable*

Інтерфейс Runnable повинен бути реалізований будь-яким класом, який буде ніціалізіровать окремий потік виконання.

Runnable визначає тільки один абстрактний метод - **run** (), Який є точкою входу в потік. Він визначений так:

- abstract void run()

Створювані вами потоки повинні реалізувати даний метод.

# Клас *Thread*

Клас Thread створює новий потік виконання. В ньому визначено такі конструктори:

- Thread ()
- Thread (Runnable *threadOb)*
- Thread (Runnable *threadOb, StringthreadName)*
- Thread (String *threadName)*
- Thread (ThreadGroup *groupOb,* Runnable threadOb)
- Thread (ThreadGroup *groupOb,* Runnable *threadOb,* String *threadName)*
- Thread (ThreadGroup *groupOb,* String *threadName)*

де threadOb - Екземпляр класу, який реалізує інтерфейс Runnable і визначає, де буде починатися виконання потоку.

Ім'я потоку передається через *threadName.*Якщо ім'я не вказується, то його створює JVM (Віртуальна Java-Машина), groupOb позначає групу, до якої буде належати новий потік. Коли потокова група не визначена, новий потік належить тій же самій групі, що й породжує потік.

# Клас *ThreadGroup*

ThreadGroup створює групу потоків. У ньому визначається два конструктора:

- ThreadGroup (String groupName)
- ThreadGroup (ThreadGroup *parentOb,* String *groupName)*

Для обох форм **groupName** вказує назву групи потоків.

Перша версія створює нову групу, яка має поточний потік в якості її батька.

У другій формі батько вказується через параметр ***parentOb.***

# Класи *ThreadLocal* і*InheritableThreadLocal*

У Java 2 до пакету java.lang додані два нових класу, пов'язаних з потоками:

ThreadLocal. Використовується для створення потокових локальних змінних. Кожен потік буде мати свою власну копію потокової локальної змінної.

InheritableThreadLocal. Створює потокові локальні змінні, які можна успадковувати.

# Клас *Package*

У Java 2 доданий клас з ім'ям Package, Який інкапсулює дані, пов'язані з версією пакета.

Інформація про версію пакету стає все більш важливою через швидке зростання і спеціалізації пакетів і тому, що Java-Програма повинна знати, яка версія пакету є доступною.

# Клас *SecurityManager*

SecurityManager - Це абстрактний клас, який ваші підкласи можуть реалізувати для створення менеджера безпеки.

В загальному випадку ви не повинні реалізовувати власний менеджер безпеки.

Однак якщо ви це робите, то повинні звернутися до документації, яка поставляється з вашою системою розробки Java-Програм.

# Інтерфейс *Comparable*

У Java 2 до java.iang доданий новий інтерфейс- Comparable.

Об'єкти класів, які реалізують Comparable, Можуть бути впорядкованими.

Іншими словами, класи, що реалізують Comparable, Містять об'єкти, які можна порівнювати деяким значущим способом.

Інтерфейс Comparable оголошує єдиний метод, призначений для визначення того, що Java 2 називає *природним впорядкування* примірників (об'єктів) класу. Сигнатура цього методу:

- int compareTo(Object *obj)*

Метод порівнює викликає об'єкт з *obj.*Він повертає 0, якщо значення рівні; негативне значення, якщо викликає об'єкт менше параметра. У будь-якому іншому випадку повертається позитивне значення.

# (Під)Пакет *java.lang.ref*

Засоби складання "сміття" в Java автоматично визначають, коли посилання до об'єкта відсутні.

У цьому випадку передбачається, що об'єкт більше не потрібен, і його пам'ять регенерується (відновлюється).

Класи в пакеті java.iang.ref, Які були додані в Java 2, забезпечують більш гнучкий контроль над процесом складання "сміття".

Наприклад, припустимо, що ваша програма створила численні об'єкти, які ви хочете багаторазово використовувати в більш пізній час. Ви можете продовжувати підтримувати посилання до цих об'єктів, але це може зажадати дуже багато пам'яті.

Замість цього можна визначити "м'які" посилання до об'єктів. Об'єкт, який є "м'яко доступним", може бути відновлений сборшіком сміття, якщо достатньо нижній пам'яті. В цьому випадку збирач сміття встановлює "м'які" посилання до того об'єкту на null. В іншому випадку він зберігає об'єкт для можливого майбутнього використання.

# (Під) Пакет java.lang.reflect

*Відображення* (reflection) - Це здатність програми анатізіровать саму себе.

Пакет java.lang.reflect забезпечує здатність отримати інформацію про поля, конструкторах, методах і модифікатори класу.

Ви потребуєте цієї інформації, щоб будувати програмні інструментальні засоби, які дають можливість працювати з компонентами Java Beans. Інструментальні засоби використовують відображення, щоб визначати динамічно характеристики компонента.

Крім того, пакет java.lang.reflect включає клас, який дає можливість створювати і звертатися з масивами *динамічно.*

# Пакет *java.util:* структура колекцій

*Колекція* - це група об'єктів.

Додавання колекцій викликало фундаментальні зміни в структурі та архітектури багатьох елементів пакета java.util.

Вони також розширили коло завдань, до яких може застосовуватися пакет.

# Короткий огляд колекцій

*Структура колекцій* (collections framework)Java стандартизує спосіб, за допомогою якого ваші програми обробляють *групи об'єктів.*

Структура колекцій була розроблена для кількох цілей.

- По-перше, структура повинна була бути високоефективною. Дійсно, реалізації фундаментальних колекцією (динамічних масивів, пов'язаних списків, дерев і хеш-таблиць) в структурі колекцій Java 2 високо ефективні. Вам рідко (а може і ніколи) потрібно кодувати одну з цих "машин даних" вручну.
- По-друге, структура колекцій повинна була дозволити різним типам колекцій працювати схожим один на одного чином і з високим ступенем здатності до взаємодії.
- По-третє, розширення та / або адаптація колекції повинна була бути простою.

На довершення до всього, повна структура колекцією розроблена в оточенні набору стандартних інтерфейсів. Ви також можете реалізувати власну колекцію за вашим вибором.

# Короткий огляд колекцій

- *Алгоритми* - інша важлива частина механізму колекцій. Алгоритми працюють на колекціях і визначені як статичні методи в класі Collections. Таким чином, вони доступні для всіх колекцій. Алгоритми забезпечують стандартні засоби маніпулювання колекціями.
- *Ітератор* забезпечує універсальний, стандартизований спосіб доступу до елементів колекції - по одному. Таким чином, ітератор забезпечує кошти *перерахування вмісту колекції.* Тому (і тільки з дрібними змінами) код, який циклічно проходить, скажімо, через *набір,* може також використовуватися для циклічного проходу *списку,* наприклад.
- *Карта відображенні* (map) Зберігає пари ключ / значення. Хоча карти (відображень) - не "колекції",вони повністю інтегровані з колекціями. Мовою структури колекцій, ви можете отримати *колекційний вид* або *подання* (collection-view) Карти відображенні. Подібне уявлення містить елементи карти відображень, що зберігаються у вигляді колекції. Таким чином, ви можете обробляти вміст відображення як колекцію.
- Механізм колекцій був пристосований до деяких з початкових класів, визначених у java.util так, щоб вони також могли бути інтегровані в нову систему.
- Технологія колекцій Java подібна (по духу) Стандартної Бібліотеці Шаблонів (STL,Standard Template Library), Визначеної в C + +. Те, що C + + називає *контейнером,* Java називає *колекцією.*

# Інтерфейси колекцій

У структурі колекцій визначено декілька інтерфейсів.

Конкретні класи просто є різними реалізаціями стандартних інтерфейсів.

# Інтерфейс *Collection*

- Об'єкти додаються в колекцію викликом методу add(). Наприклад, колекція не може безпосередньо зберігати значення типу int,char,double і т. д.. Можна додати повне вміст однієї колекції до іншої, викликаючи метод addAll ().
- Для видалення об'єкта служить метод remove (). Щоб видалити групу об'єктів, викликайте метод removeAll ().
- Викликаючи метод retainAll () можна видалити всі елементи, крім певної їх групи.
- Для чищення колекції викличте метод clear ().
- Існує можливість визначати, чи містить колекція певний об'єкт за допомогою виклику методу contains (). Ч
- об з'ясувати, чи містить одна колекція всі члени іншого, викликають метод containsAll ().
- Викликаючи метод isEmpty (), можна визначити, чи є колекція порожньою.
- Кількість елементів, що містяться в даний час в колекції, можна обчислити за допомогою методу size ().

# Інтерфейс *List*

Інтерфейс List розширює Collection і оголошує поведінку колекції, яка зберігає послідовність елементів.

Елементи можуть бути вставлені або витягнуті з допомогою їхніх позицій у списку через відлічуваний від нуля індекс.

Список може містити дубльовані елементи.

# Інтерфейс *Set*

Інтерфейс Set розширює інтерфейс Collection і оголошує поведінку колекції, не допускає дублювання елементів.

Тому метод add () повертає false, Якщо здійснюється спроба додати в набір дублюючі елементи.

У Set не визначається жодних додаткових власних методів.

# Інтерфейс *SortedSet*

Інтерфейс SortedSet розширює Set і оголошує поведінка набору, відсортованого в зростаючому порядку.

# Клас *ArrayList*

Клас ArrayList розширює AbstractList І реалізує інтерфейс List.

ArrayList підтримує динамічні масиви, які можуть рости в міру необхідності.

# Клас *LinkedList*

Клас LinkedList розширює AbstractSequentiaiList і реалізує інтерфейс List. Він забезпечує структуру даних зв'язного списку і має два наступних конструктора:

- **LinkedList ()**
- **LinkedList(Collection c)**

Перший конструктор будує порожній зв'язний список, а другий створює зв'язний список, ініціалізований елементами колекції ***c***.

# Клас *HashSet*

- Клас HashSet розширює AbstractSet і реалізує інтерфейс Set. Він створює колекцію, яка використовує хеш-таблицю для зберігання колекцій. Як
- При *хешування* інформаційний зміст ключа використовується, щоб визначити унікальне значення, зване його хеш-кодом.
- Потім хеш-код застосовується як індекс (номер) елемента, в якому зберігаються дані, пов'язані з ключем.
- Перетворення ключа в його хеш-код виконується автоматично - ви ніколи не бачите сам хеш-код.

# Клас *TreeSet*

Клас TreeSet забезпечує реалізацію інтерфейсу Set і використовує ієрархічну (деревоподібну) структуру для зберігання даних.

Об'єкти зберігаються в сортувати за зростанням порядку.

У класі визначено такі конструктори:

- **TreeSet ()**
- **TreeSet (Collection з)**
- **TreeSet (Comparator *camp)***
- **TreeSet (SortedSet ss)**

- Перша форма створює порожній деревовидний набір, який буде сортуватися у висхідному порядку згідно природному порядку його елементів.
- Друга - формує деревовидний набір, який містить елементи колекції с.
- Третя форма створює порожній деревовидний набір, який буде сортуватися відповідно до компаратором *співр.*
- Четверта - будує деревовидний набір, який містить елементи сортованого набору *ss.*

# Доступ до колекції через ітератор

Часто потрібно циклічно проходити елементи в колекції, наприклад, для відображення їх на екрані.

Найпростіший спосіб зробити це - використовувати *ітератор* (iterator), Тобто спеціальний об'єкт, який реалізує один з інтерфейсів – або Iterator, або ListIterator.

Інтерфейс Iterator дає можливість циклічно пройти через колекцію, отримуючи або видаляючи її елементи.

Інтерфейс ListIterator розширює iterator, Забезпечуючи двонаправлений обхід списку, і модифікацію елементів.

# Робота з картами відображень

На додаток до колекцій в Java 2 до java.util додаються карти відображень.

*Карта відображень* (map) - Це об'єкт, який зберігає асоціації (зв'язку) між ключами і значеннями, або пари ключ / значення.

По заданому ключу ви можете знайти його значення.

І ключі і значення є об'єктами.

Ключі повинні бути унікальними, але значення можуть бути дубльованими.

Одні карти (відображень) допускають null-Ключі і null-Значення, інші - ні.

# Компаратори

- Як TreeSet, Так і TreeMap зберігають елементи в відсортованому вигляді.
- Однак точне визначення порядку сортування виконує *компаратор.*
- За замовчуванням, ці класи зберігають свої елементи в порядку, який Java називає "природним" (natural). Це впорядкування за зростанням значення, яке ви зазвичай і очікуєте (A перед B, 1 перед 2, і т. д.).
- Якщо ви хочете упорядкувати елементи іншим способом, то вкажіть (в аргументі конструктора) об'єкт типу comparator, Коли створюєте набір або карту.
- Це забезпечить можливість точно управляти сортуванням елементів, що зберігаються в сортованих колекціях і картах.

# Алгоритми колекцій

Структура колекцій містить кілька алгоритмів, які можуть застосовуватися до колекцій і картам відображень.

Ці алгоритми визначені як статичні методи в класі Collections.

# Масиви

У Java 2 до java.util доданий новий клас з ім'ям Arrays.

Він забезпечує різні методи, які корисні при роботі з масивами.

Хоча ці методи технічно не є частиною структури колекцій, вони допомагають заповнити прогалину між колекціями та масивами.

# Інтерфейс *Enumeration*

Інтерфейс Enumeration визначає методи, за допомогою яких ви можете *перерахувати* (Отримати по одному) елементи колекції об'єктів.

Цей успадкований інтерфейс набагато поступається Iterator-Інтерфейсу колекцій.

Enumeration розглядається застарілим для нового коду, хоча і не виключено з пакета.

Однак він використовується деякими методами успадкованих класів (таких як Vector iProperties), Іншими API-Класами і в даний час широко застосовується у прикладних кодах.

# Клас *Vector*

Клас Vector реалізує динамічний масив.

Він подібний до класу ArrayList, Але з двома відмінностями: vector синхронізований і містить багато успадкованих методів, які не є частиною структури колекцій.

З випуском Java 2 клас Vector був перепроектований так, щоб розширити клас AbstractList і реалізувати інтерфейс List.

# Клас *Stack*

Клас Stack є підкласом vector-Класу, який реалізує стандартний стек LIFO.

В Stack визначений лише один (задається за замовчуванням) конструктор, який створює порожній стек, Stack включає всі методи, визначені в класі vector, І додає кілька власних.

# Клас *Dictionary*

Dictionary - Абстрактний клас, який представляє архів для зберігання даних типу ключ / значення і працює багато в чому аналогічно класу Map.

Задаючи ключі, ви можете зберігати значення в об'єкті класу Dictionary.

Як тільки значення збережено, ви можете відшукати й витягти його, використовуючи ключ.

Таким чином, подібно карті відображень, словник можна уявляти собі як список пар ключ / значення.

# Клас *Hashtable*

Клас Hashtable був частиною вихідного пакета java.util і є конкретною реалізацією Dictionary.

Однак в Java 2Hashtable перепроектований так, що він реалізує ще й інтерфейс Map.

Таким чином, Hashtable тепер інтегрований в структуру колекцій.

Він подібний до класу HashMap, Але синхронізований.

# Клас *Properties*

Properties - Це підклас класу Hashtable.

Він використовується для підтримки списків значень, в яких ключ і значення є string-Об'єктами.

Клас Properties використовується багатьма іншими класами Java.

# Використання методів *store ()* і*load ()*

Одним з найбільш корисних властивостей класу Properties є можливість збереження (на диску) і завантаження (з диска) інформації, що міститься в Properties-Об'єкті, за допомогою методів store () і load().

Ви в будь-який час можете записати Properties-Об'єкт в потік або вважати його назад.

Це робить списки властивостей особливо зручними для реалізації простих баз даних.

# Клас *StringTokenizer*

Обробка тексту часто складається з синтаксичного аналізу відформатованої вхідного рядка.

*Синтаксичний аналіз* (Parsing) - Це поділ тексту на безліч дискретних частин або *лексем* (Tokens), які в деякій послідовності можуть передавати семантичне значення.

Клас StringTokenizer, часто званий *лексичним аналізатором* або *сканером,* забезпечує перший крок в процесі такого аналізу. Він реалізує інтерфейс Enumeration.

# Клас *BitSet*

Клас Bitset створює спеціальний тип масиву, який містить бітові значення.

Масив може збільшуватися в розмірі, наскільки потрібно. І це робить його схожим на бітовий вектор.

# Клас *Date*

Клас Date інкапсулює поточну дату і час.

Клас Date підтримує наступні конструктори:

- **Date ()**
- **Date(long *millisec)***

Перший конструктор ініціалізує об'єкт з поточною датою і часом.

Другий приймає один аргумент, який дорівнює числу мілісекунд, які минули з півночі 1 січня 1970

# Клас *Calendar*

Абстрактний клас Calendar забезпечує набір методів, який дозволяє перетворити час у мілісекундах в набір таких корисних компонентів, як рік, місяць, день, година, хвилина та секунда.

Передбачається, що підкласи Calendar будуть забезпечувати певні функціональні можливості для інтерпретації інформації часу згідно з їх власними правилами.

Calendar не містить загальних (public) Конструкторів і визначає кілька захищених (protected) Екземплярність змінних:

- boolean areFieldsSet вказує, чи були встановлені компоненти часу;
- int fields[] Є масивом цілих чисел, який містить компоненти часу;
- boolean isSet[] Є масив логічних значень, який вказує, чи був встановлений певний часовий компонент;
- long time містить поточний час для цього об'єкта;
- boolean isTimeSet вказує, чи було встановлено поточний час.

# Клас *GregorianCalendar*

Клас GregorianCalendar є конкретною реалізацією абстрактного класу Calendar, Який імітує нормальний григоріанський календар.

Метод getInstance () класу Calendar повертає об'єкт типу GregorianCalendar, ініціалізований поточною датою і часом із заданими за умовчанням мовному регіоном і часового поясу.

# Клас *TimeZone*

Клас TimeZone дозволяє працювати з відліком часового поясу від середнього часу за Гринвічем (GMT,Greenwich Mean Time), Званому також *скоординованим універсальним часом* (UTC,Coordinated Universal Time) І обчислює літній час (daylight saving time).

# Клас *SimpleTimeZone*

Клас SimpleTimeZone - Зручний підклас класу TimeZone.

Він реалізує абстрактні методи класу TimeZone і дозволяє працювати з часовими поясами за григоріанським календарем.

# Клас *Locale*

Клас Locale створює об'єкти, кожен з яких описує географічний чи культурний регіон.

Це один з декількох класів, що забезпечують можливість написання програм, які можуть виконуватися в різних міжнародних середовищах.

# Клас *Random*

Клас Random є генератором псевдовипадкових чисел.

*Псевдовипадкові числа* формують рівномірно розподілені послідовності.

# Клас *Observable*

Клас Observable використовується для створення підкласів, за якими можуть спостерігати інші частини вашої програми.

Коли об'єкт такого підкласу піддається зміні, спостерігають класи повідомляються про це.

Спостерігають класи повинні реалізовувати інтерфейс Observer, Який визначає метод update ().

Коли спостерігач повідомляється про зміну в спостережуваному об'єкті, викликається метод update ().

# Пакет *java.util.zip*

Пакет java.util.zip забезпечує здатність читати і записувати файли в популярному ZIP iGZIP файлових форматах. Доступні як ZIP, Так і GZIP потоки введення і виведення. Інші класи реалізують ZLIB-Алгоритми для стиснення і декомпресії.

# Пакет *java.util.jar*

Пакет java.utii.jar забезпечує можливість читати і записувати файли JAR (Java Archive).

# Пакет *java.io*

Досліджується пакет java.io, Який забезпечує підтримку операцій введення / виводу (I/O,Input/Output).

# Клас *File*

Клас File має справу безпосередньо з файлами і файлової системою.

У класі File не вказується, як потрібно витягати або зберігати інформацію в файлах, в ньому описуються властивості самого файлу.

Для отримання або управління інформацією, пов'язаною з дисковим файлом, використовується *об'єкт* типу (класу) File.

Мова тут йде про такі властивості файла, як права доступу, час і дата створення і зміни, шлях в ієрархії підкаталогів і т. д.

# Каталоги

Каталог представляє собою File-Об'єкт, який містить список інших файлів і каталогів.

Коли ви створюєте об'єкт типу File, І він є каталогом, виклик методу isDirectory() повертає true.

В цьому випадку ви можете застосувати метод list () До вказаного об'єкту, щоб витягти з нього список інших файлів і каталогів.

# Використання інтерфейсу *FilenameFilter*

Часто потрібно обмежити кількість файлів, що повертаються методом list (), Щоб включати в список тільки ті з них, імена яких відповідають деякому зразку або фільтру.

Для цього потрібно використовувати другу форму методу list ():

- **String[] list(FilenameFilter *FFObj)***

Тут *FFobj* - Об'єкт класу, який реалізує інтерфейс типу FilenameFilter.

# Альтернативний метод *listFiles ()*

У Java 2 доданий варіант методу list () з ім'ям UstFiies (), Який теж може виявитися корисним. Сигнатури трьох (перевантажених) варіантів listFiles ():

- **File[] listFiles ()**
- **File[] listFiles(FilenameFilter *FFObj)***
- **File[] listFiles(FileFilter *FObj)***

Ці методи повертають список файлів у вигляді *масиву об'єктів* типу File (Не просто у вигляді рядків).

Перший метод повертає всі файли, а другий - тільки ті, що удовлетврряют фільтру імен *FFObj.*За винятком повернення масиву File-Об'єктів, ці дві версії listFiles () Працюють аналогічно їх еквівалентному методу list () ..

Третя версія listFiles про фільтрує файли не просто по іменах, а по їх *повним* іменам (з шляхами в ієрархії каталогів), і повертає файли з тими іменами шляхи, які задовольняють файловому фільтру *FObj.*

# Клас *Inputstream*

Inputstream - Абстрактний клас, який визначає Java-Модель потокового вводу байтів.

Всі методи даного класу в аварійних ситуаціях викидають виключення типу IOException.

# Клас *OutputStream*

OutputStream - Абстрактний клас, який визначає байтовий вихідний потік.

Всі методи в цьому класі повертають значення void і викидають в разі помилок виключення типу iOException.

# Клас *FileInputStream*

Клас FileInputStream створює inputstream-Об'єкт, який можна використовувати для читання байтів з файлу. Нижче показано два його найбільш загальних конструктора:

- **FileInputStream(String *filepath)***
- **FileInputSream(File *fileObj)***

Кожен може викидати виключення типу FileNotFoundException.

Параметри: *filepath* - Повне ім'я (зі шляхом) файла; *fileObj* -об'єкт типу (класу) File, Який описує файл.

# FileOutputStrem

Клас FileOutputStream створює об'єкт outputstream, Який можна застосовувати для запису байтів в файл. Зазвичай використовуються такі конструктори цього класу:

- **FileOutputStream(String *filePath)***
- **FileOutputStream(File *fileObj)***
- **FileOutputStream (String *filePath,* boolean *append)***

Вони можуть викидати виключення типу IOException АБО SecurityException.

Параметри: ***filePath*** - Повне ім'я шляху файлу; *fiieobj* - Об'єкт типу File, Який описує файл. Якщо **append** -true, Файл відкривається в режимі додавання.

Створення FiieOutputstream-Об'єкта не залежить від того, чи існує вже файл. FiieOutputstream-Конструктор створить файл перед його відкриттям для висновку, коли ви створюєте об'єкт. Якщо ви спробуєте відкрити файл лише для читання, буде викинуто виключення типу IOException.

# Клас *ByteArrayInputStream*

ByteArrayinputstream - Реалізація вхідного потоку, яка використовує байтовий масив як джерело. Цей клас має два конструктора, кожен з яких використовує байтовий масив в якості джерела даних:

- **ByteArrayInputStream (byte *array [* ])**
- **ByteArrayInputStream (byte *array [* ],** int ***start,*** **int numBytes)**

Тут **array** - джерело введення. Другий конструктор створює inputstream-Об'єкт, що складається з байтового подмассіва, який починається з позиції **start** і має довжину **numBytes** байтів.

# Клас *ByteArrayOutputStream*

Клас ByteArrayOutputStream реалізує вихідний потік, який використовує byte-Масив в якості пункту призначення. ByteArrayOutputStream має два конструктора в такій формі:

- **ByteArrayOutputStream()**
- **ByteArrayOutputStream (int *mxnBytes)***

У першій формі створюється 32-байтним буфер.

У другій - буфер з розміром, рівним вказаною в параметрі *nwnBytes.*Буфер міститься в захищеному полі buf класу ByteArrayOutputStream. При необхідності розмір буфера збільшується автоматично. Число байтів буфера міститься в захищеному полі count класу ByteArrayOutputStream.

# Фільтровані байтові потоки

*Фільтровані потоки* - Просто оболонки (wrappers) Навколо основних потоків введення або виведення, які прозоро забезпечують деякий розширений рівень функціональних можливостей.

Доступ до цих потокам зазвичай виконується за допомогою методів, які використовують породжує потік, пов'язаний з суперкласом фільтрованих потоків.

Типові застосування фільтрованих потоків - буферизація, трансляція символів і необроблених даних.

Їх конструктори мають форму:

- **FilterOutputStream (Outputstream *os)***
- **FilterInputStream (Inputstream *is)***

# Буферізовані байтові потоки

Байтовий *буферізванний потік* розширює клас фільтрованого потоку, приєднуючи буфер пам'яті до потоків введення / виведення.

Такий буфер дозволяє виконувати I/ O-операції не з одним, а з декількома байтами одночасно, і, отже, збільшує ефективність роботи програми.

# Клас *BufferedInputStream*

Клас BufferedInputStream дозволяє "обгорнути" будь InputStream-Об'єкт в буферізірованний потік і досягти тим самим поліпшення ефективності.

BufferedInputStream містить два конструктора:

- **BufferedInputStream (Inputstream inputstream)**
- **BufferedInputStream (Inputstream inputstream, int bufSize)**

Перша форма створює буферізірованний потік, використовуючи розмір буфера, заданий за замовчуванням.

У другій формі розмір буфера передається в параметрі *bufsize.* Використання розмірів, кратних сторінці пам'яті, дискового блоку і т. д. може мати істотний позитивний вплив на ефективність. Це, однак, залежить від реалізації. Оптимальний розмір буфера в загальному випадку залежить від операційної системи, обсягу наявної пам'яті і конфігурації хост-машини. Найкраще розмір буфера мати кратним 8192 байтам, але і приєднання навіть досить маленького буфера до потоку вводу / виводу - завжди хороша ідея.

# Клас *BufferedOutputStream*

Клас BufferedOutputStream подібний будь класу Outputstream-Ієрархії за винятком того, що доданий метод flush ().

Його використання гарантує, що буфери даних фізично записуються на актуальне вихідний пристрій.

# Клас *PushbackInputStream*

Одне з нових застосувань буферизації - виконання повернення ліченого байта назад у вхідний потік (операція pushback). Клас PushbackInputStream здійснює цю ідею. Він забезпечує механізм, що дозволяє "швидко поглянути "на те, що відбувається у вхідному потоці без його переривання. Pushbackinputstream має такі конструктори:

- **Pushbackinputstream (InputStream *inputstream)***
- **Pushbackinputstream (InputStream *inputstream,* int *numBytes)***

Перша форма створює потоковий об'єкт, який дозволяє повернути один байт у вхідний потік.

Друга форма створює потік, який має спеціальний pushback-Буфер довжиною ***numBytes*** байтів. Він дає можливість виконувати множинний повернення байтів у вхідний потік.

# Клас *SequenceInputStream*

Клас SequenceInputStream дозволяє зчіплювати безліч InputStreams-Об'єктів.

Конструкція SequenceInputStream відрізняється від будь-якого іншого класу InputStream-Ієрархії.

Конструктор SequenceInputStream використовує в якості параметрів або пару inputstream-Об'єктів або Enumeration-Об'єкт з inputstream-Компонентами:

- **SequenceInputStream (Inputstream *first,* Inputstream *second)***
- **SequenceInputStream (Enumeration streamEnum)**

# Клас *Printstream*

Клас Printstream забезпечує всі можливості форматування. Printstream має два конструктора:

- **PrintStream(OutputStream *outputStream)***
- **PrintStream (OutputStream outputStream, boolean *flushOnNewline)***

де ***fiushOnNewiine*** управляє скиданням вихідного потоку кожен раз, коли зустрічається newline-Послідовність (\n). Якщо *fiushOnNewiine* -true, Скидання виконується автоматично, якщо - false, Скидання не автоматичне. Перший конструктор виконує неавтоматичний скидання.

# Клас *RandomAccessFile*

Клас RandomAccessFile інкапсулює файл прямого доступу. Він не є похідним від класів InputStream або OutputStream. Замість цього, реалізує інтерфейси DataInput iDataOutput, Які визначають базові методи введення / виведення. Він також підтримує запити позиціонування - тобто ви можете встановлювати всередині файлу *покажчик* (Файлу). Клас володіє двома конструкторами:

- **RandomAccessFiie (File *fileObj,* String *access)* throws IOException**
- **RandomAccessFiie (Stringy filename, String *access)* throws IOException**

У першій формі параметр ***fileObj*** визначає ім'я файлу, що відкривається як File-Об'єкт.

У другій формі ім'я файлу передається через параметр ***filename.*** В обох **випадках *access*** визначає, який тип доступу дозволяється до файлу. Якщо (у виклику конструктора) вказано значення "r", То файл можна читати, але не записувати, якщо -"rw", То файл відкритий в режимі читання-запису.

# Символьні потоки

Хоча класи байтових потоків забезпечують достатні функціональні можливості для обробки будь-якого типу операцій введення / виводу, вони не можуть працювати безпосередньо з Unicode-Символами.

Так як однієї з головних цілей Java була підтримка філософії "писати якось, виконувати скрізь", виникла необхідність включити пряму підтримку введення / виводу для символів.

# Клас *Reader*

Reader - Абстрактний клас, який визначає Java-Модель поточного символьного введення.

# Клас *Writer*

Writer - Абстрактний клас, який визначає Java-Модель поточного символьного виводу.

Всі методи цього класу повертають значення void і викидають виключення типу ioException у разі помилок.

# Клас *FileReader*

Клас FileReader створює Reader-Об'єкт, який можна використовувати для читання вмісту файлу. Нижче показано два його зазвичай використовуються конструктора:

- **FileReader(String *filePath)***
- **FileReader(File *fileObj)***

Будь-хто може викидати виняток типу FileNotFoundException. Тут *filePath* -повне ім'я файлу; *fileObj* - Об'єкт класу File, Який описує файл.

# Клас *FileWriter*

Filewriter створює writer-Об'єкт, який можна використовувати для запису файлу. Його зазвичай використовуються конструктори мають наступні сигнатури:

- **FileWriter (String *filePath)***
- **FileWriter (String *filePath,* boolean *append)***
- **FileWriter (File *fileObj)***

Вони можуть викидати виключення типу IOException АБО SecurityException.
Тут **filePath** - Повне ім'я файлу; *fiieobj* -File-Об'єкт, який описує файл. Якщо ***append*** -true, То висновок додається в кінець файлу.

# Клас *CharArrayReader*

CharArrayReader є реалізацією вхідного потоку, яка використовує символьний масив як джерело. Даний клас має два конструктора, кожен з яких отримує символьний масив в якості джерела даних:

- **CharArrayReader (char array [])**
- **CharArrayReader (char [], int start,** int ***nxunChars)***

де **array[]** - Джерело введення. Другий конструктор створює Reader-Об'єкт з підмножини вихідного символьного масиву, який починається з індексного позиції ***start*** **(**у вихідному масиві) і має довжину ***numChars*** символів.

# Клас *CharArrayWriter*

CharArrayWriter - Реалізація вихідного потоку, яка записує дані в символьний масив. CharArrayWriter має два конструктора:

- **CharArrayWriter()**
- **CharArrayWriter (int *numchars)***

У першій формі створюється буфер з розміром, заданим за умовчанням.

У другій - буфер з розміром, зазначеним у параметрі *numchars.*Буфер міститься в поле buf класу CharArrayWriter. Якщо необхідно, розмір буфера буде збільшено автоматично. Число символів, що зберігаються в буфері, міститься в поле count класу CharArrayWriter.

# Клас *BufferedReader*

BufferedReader покращує ефективність, буферизуючи введення. Він містить два конструктора:

- **BufferedReader (Reader *inputstrearn)***
- **BufferedReader (Reader *inputstream,* int *bufSize)***

Перша форма створює буферізірованний символьний потік, використовуючи розмір буфера, заданий за замовчуванням.

У другій формі розмір буфера передається конструктору через параметр ***bufSize.***

# Клас *BufferedWriter*

Bufferedwriter - Це клас writer, До якого додано метод flush (), Який гарантує, що буфери даних фізично записуються в актуальний вихідний потік. Використання BufferedWriter може збільшити ефективність, зменшуючи кількість фактичних записів даних у вихідний потік. BufferedWriter має два конструктора:

- **BufferedWriter (Writer *outputstream)***
- **BufferedWriter (Writer *outputstream,* int *bufSize)***

Перша форма створює буферізірованний потік, використовуючи буфер з розміром, заданим за умовчанням.

У другій формі розмір буфера передається через параметр ***bufSize.***

# Клас *PushbackReader*

Клас PushbackReader дозволяє повертати у вхідний потік один або кілька символів. Це дає можливість "заглянути вперед" у вхідний потік. Клас містить два конструктора:

- **PushbackReader (Reader *inputStream)***
- **PushbackReader (Reader inputStream, int *bufSize)***

Перша форма створює буферізірованний потік, що дозволяє повернути назад один символ.

У другій формі розмір повертається порції даних вказується параметром ***bufSize.***

# Клас *PrintWriter*

Printwriter - По суті, символьна версія класу Printstream. Він визначає методи форматного виведення print () і println ().Printwriter має чотири конструктора:

- **PrintWriter (OutputStream outputStream)**
- **PrintWriter (OutputStream OutputStream, boolean *flushOnNewlIne)***
- **PrintWriter (Writer *outputStream)***
- **PrintWriter (Writer outputStream, boolean *flushOnNewline)***

де *fiushOnNewiine* управляє скиданням вихідного потоку кожен раз, коли виводиться символ нового рядка newline (\n).

Якщо flushOnNewline -true, Скидання виконується автоматично, якщо - false, Скидання не автоматичне.

Перший і третій конструктори організують неавтоматичний скидання.

# Серіалізация

*Серіалізсщія* (serialization) - Це процес запису стану об'єкта у формі байтового потоку. Це корисно, коли ви хочете зберегти стан своєї програми в постійній області пам'яті, наприклад, у файлі.

Пізніше ви можете відновити ці об'єкти, використовуючи процес *десеріалізациі* (deserialization).

Серіалізация необхідна також для реалізації механізму виклику вилучених методів (RMI,Remote Method Invocation). RMI дозволяє Java-Об'єкту на одній машині викликати метод Java-Об'єкту, що знаходиться на віддаленій машині. Віддалений об'єкт можна вказувати як аргумент віддаленого методу. Посилає машина серіалізірует об'єкт і передає його. Приймаюча машина десеріалізует прийнятий об'єкт.

# Інтерфейс *Serializable*

Засоби сериализации можуть працювати тільки з об'єктами, які реалізують інтерфейс Serializable.

Інтерфейс Serializable не визначає жодних членів. Він просто вказує на те, що клас може бути серіалізовать (тобто перетворений у послідовну форму). Якщо клас - серіалізовать (тобто реалізує інтерфейс serializable), То все його підкласи також серіа-лізованних.

Змінні, які оголошені як transient або static, Не зберігаються засобами сериализации.

# Інтерфейс *Externailzable*

Засоби сериализации і десеріалізациі Java були розроблені так, щоб більша частина роботи по збереженню та відновленню стану об'єкта виконувалася автоматично.

Проте є ситуації, в яких програмісту потрібно мати контроль над цими процесами. Наприклад, якщо бажано використовувати техніку стиску або криптографічного кодування. Саме для таких ситуацій і розроблений інтерфейс Externaiizabie. В ньому визначаються два методи:

- **void readExternal (Objectinput inStream) throws IOException, ClassNotFoundException**
- **void writeExternal (ObjectOutput outStream) throws IOException**

Параметри цих методів: ***inStream*** - Байтовий потік, з якого об'єкт повинен бути лічений; **outStream** - Байтовий потік, в який об'єкт повинен бути записаний**.**

# Інтерфейс *ObjectOutput*

Інтерфейс ObjectOutput розширює інтерфейс Dataoutput і підтримує сериализацию об'єктів

# Клас *ObjectOutputStream*

Клас ObjectOutputStream розширює інтерфейс outputstream і реалізує інтерфейс objectoutput. Він відповідає за запис *об'єктів* в потік. Конструктор цього класу:

- **ObjectOutputStream(Outputstream *outStream)*throws IOException**

де параметр *outstream* - Вихідний потік, в який будуть записуватися серіалізовать об'єкти.

# Інтерфейс *ObjectInput*

Інтерфейс ObjectInput розширює інтерфейс Datainput.

Він підтримує серіалізацію об'єктів.

# Клас *ObjectInputStream*

Клас ObjectInputStream розширює клас inputstream і реалізує інтерфейс objectInput.ObjectInputStream відповідає за читання об'єктів з вхідного потоку. Конструктор цього класу:

- **ObjectInputStream (Inputstream *inStream)* throws IOException, StreamCorruptedException**

Параметр *inStream* - Вхідний потік, з якого повинні зчитуватися серіалізовать об'єкти.

# Переваги потоків

Потоковий інтерфейс вводу / виводу в Java забезпечує чітку абстракцію для складних і часто громіздких задач.

Композиція фільтрованих потокових класів дозволяє динамічно формувати замовний потоковий інтерфейс, що задовольняє ваші вимоги до передачі даних.

Очікується, що в майбутньому все більш і більш важливу роль в Java-Програмуванні гратиме сериализация об'єктів.

Java-Класи серіалізовать введення / виводу забезпечують мобільний (переносний) рішення цієї досить складної задачі.

# Дякую за увагу!