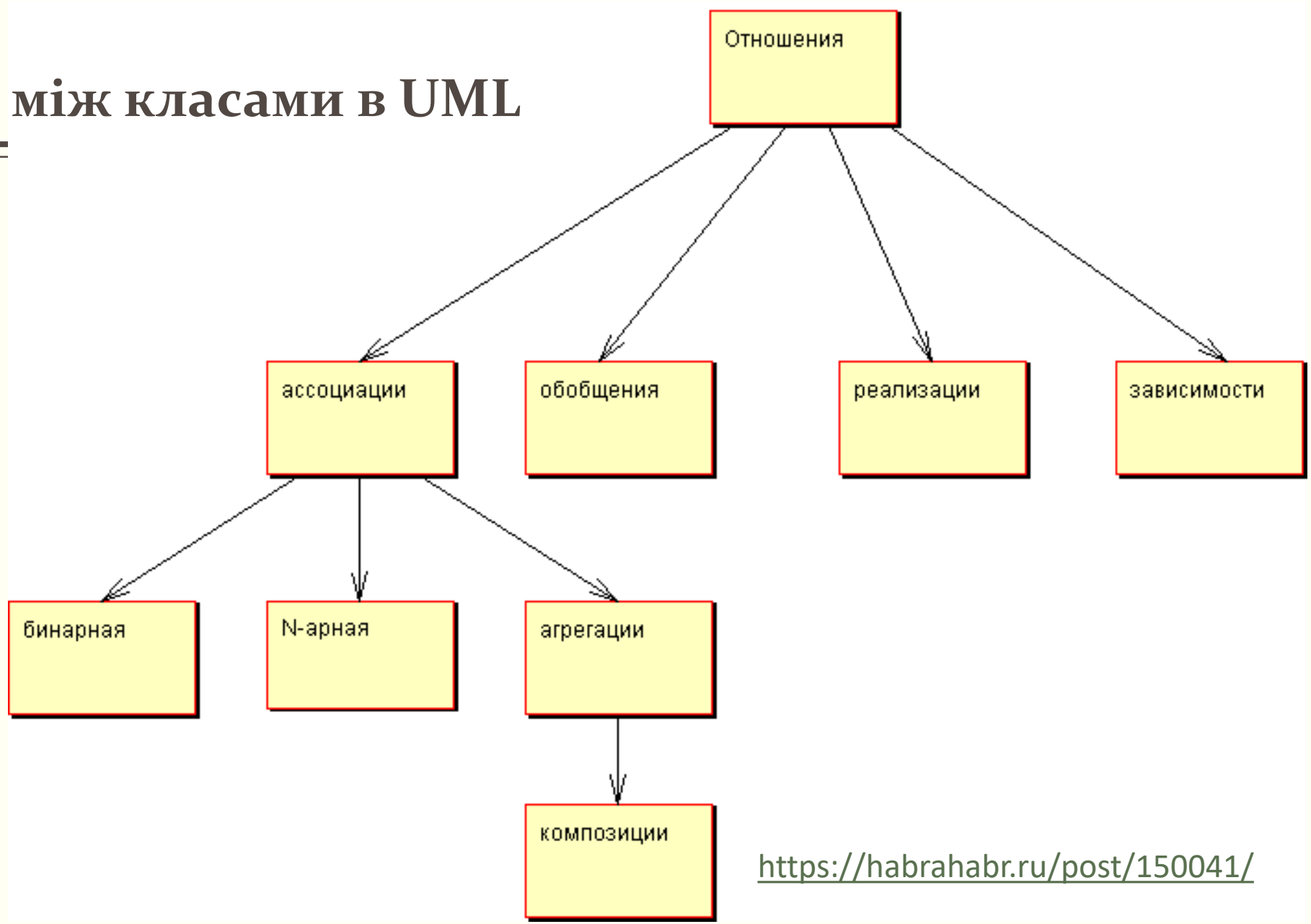
ОГЛЯД ВІДНОШЕНЬ МІЖ КЛАСАМИ. ГЕНЕРАЛІЗАЦІЯ

Питання 2.5.

Відношення між класами в UML



https://habrahabr.ru/post/150041/

Узагальнення (наслідування). Створення ієрархії класів

- Людина любить категоріювати предмети: автомобіль є транспортним засобом, накопичувальний рахунок відноситься до банківських рахунків і т. д.
  - З точки зору програмування автомобілі *наслідують* (inherit) **стан** транспортного засобу (як виробник і колір) та його **поведінку** (виконання парковки, відображення пройденого шляху та ін.)
  - Накопичувальний рахунок наслідує стан банківського рахунку (як баланс) та його поведінку (здійснити внесок, зняти гроші тощо)
- Наслідування – ієрархічна залежність між схожими категоріями сутностей, у яких одна категорія успадковує стан та поведінку принаймні від однієї іншої категорії сутностей.
  - Наслідування від однієї категорії – одиничне (single inheritence),
  - від мінімум двох категорій – множинне (multiple inheritence)

Одиничне та множинне наслідування

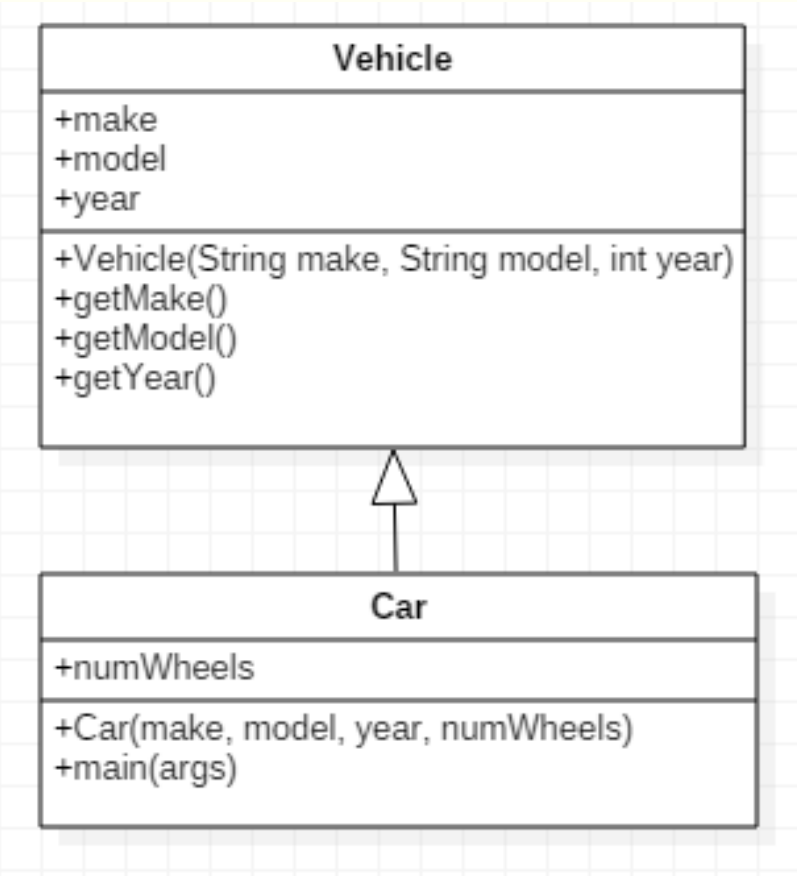
- Java підтримує одиничне наслідування в контексті класу
  - Повторне використання коду: навіщо заново винаходити велосипед?
  - Клас наслідує стан та поведінку від іншого класу за допомогою механізму ***розширення*** (class extention)
  - Оскільки задіяно класи, такий тип вважається ***наслідуванням реалізації*** (implementation inheritance)
- В контексті інтерфейсу Java підтримує як одиничне, так і множинне наслідування
  - Клас наслідує шаблони поведінки (behavior templates) від одного або більше інтерфейсів за допомогою реалізації інтерфейсу (interface implementation)
  - Інтерфейс наслідує шаблони поведінки від одного або більше інтерфейсів за допомогою розширення інтерфейсу (interface extension)
  - Оскільки задіяно інтерфейси, такий тип вважається ***наслідуванням інтерфейсу*** (*interface inheritance*)
- Повторне використання коду відбувається за допомогою ретельного розширення класів, реалізації та розширення інтерфейсів.
  - Ви починаєте з чогось близького до того, що бажаєте, а потім розширюєте його для досягнення своїх цілей.
  - Ви не використовуєте код повторно простим копі-пастом. Такий код буде надмірним (тобто non-reusable) та забагованим.

Розширення класів

- Використовується зарезервоване слово **extends**.
  - Якщо Ви хочете, щоб один з Ваших класів не могли розширювати (з питань безпеки чи інших причин), необхідно оголосити клас як final: final class Vehicle {...}

```
class Vehicle
{
   // member declarations
}

class Car extends Vehicle
{
   // member declarations
}
```



Код відображає відношення «is-a»: автомобіль **є** видом транспортного засобу:

- Клас Vehicle називають *base class, parent class, superclass*
- Клас Car називають *derived class, child class, subclass*

```
class Vehicle
{
   private String make;
   private String model;
   private int year;

   Vehicle(String make, String model, int year)
   {
      this.make = make;
      this.model = model;
      this.year = year;
   }

   String getMake()
   {
      return make;
   }

   String getModel()
   {
      return model;
   }

   int getYear()
   {
      return year;
   }
}
```

```
public class Car extends Vehicle
{
   private int numWheels;

   Car(String make, String model, int year, int numWheels)
   {
      super(make, model, year);
      this.numWheels = numWheels;
   }

   public static void main(String[] args)
   {
      Car car = new Car("Ford", "Fiesta", 2009, 4);
      System.out.println("Make = " + car.getMake());
      System.out.println("Model = " + car.getModel());
      System.out.println("Year = " + car.getYear());
      // Normally, you cannot access a private field via an object
      // reference. However, numWheels is being accessed from
      // within a method (main()) that is part of the Car class.
      System.out.println("Number of wheels = " + car.numWheels);
   }
}
```

Підклас Car має приватне поле numWheels конструктор, який ініціалізує об'єкти Car та метод main() для тестування.

Зарезервоване слово super використовується для виклику конструктора класу Vehicle.

Виклик super() аналогічний this(), проте звертається до суперкласу

Зверніть увагу!

- Виклик методу super() може відбуватись лише в конструкторі.
  - Це має бути перший код у тілі конструктора.
  - Якщо super() не задано, а суперклас не має безаргументного конструктора, компілятор повідомить про помилку, оскільки конструктор підкласу має викликати безаргументний конструктор суперкласу, якщо немає super().
- Метод main() класу Car створює об'єкт типу Car, ініціалізуючи його поля конкретними значеннями make, model, year, та number of wheels.
  - Хоча прямий доступ до поля екземпляру зазвичай не є хорошою ідеєю (порушується інкапсуляція), метод main() класу Car, що забезпечує цей доступ, присутній лише для тестування і не буде відсутнім у реальній програмі, яка буде використовувати цей клас.

```
Make = Ford
Model = Fiesta
Year = 2009
Number of wheels = 4
```

Клас, чиї екземпляри не можуть бути зміненими, називають ***незмінюваним (immutable)***.

Прикладом є клас Vehicle.

Якби не було методу main() класу Car, який може напряму зчитувати або записувати значення numWheels, клас Car теж був би незмінним.

Також клас не може наслідувати конструктори та приватні поля і методи.

Клас Car не наслідує ні конструктор Vehicle, ні його приватні поля make, model, year.

Переозначення (override) – заміна реалізації успадкованого методу в підкласі

```
class Vehicle
{
   private String make;
   private String model;
   private int year;

   Vehicle(String make, String model, int year)
   {
      this.make = make;
      this.model = model;
      this.year = year;
   }

   void describe()
   {
      System.out.println(year + " " + make + " " + model);
   }
}
```

```
public class Car extends Vehicle
{
   private int numWheels;

   Car(String make, String model, int year, int numWheels)
   {
      super(make, model, year);
   }

   void describe()
   {
      System.out.print("This car is a "); // Print without newline
      super.describe();
   }

   public static void main(String[] args)
   {
      Car car = new Car("Ford", "Fiesta", 2009, 4);
      car.describe();
   }
}
```

Метод describe() переозначає варіант з класу Vehicle для виводу опису автомобіля

Переозначення

- Виклик методу суперкласу з переозначеного методу підкласу відбувається шляхом дописування перед назвою методу «super.».
  - Якщо цього не зробити, Ви зациклитесь рекурсивним викликом переозначеного методу підкласу.
  - Використовуйте «super.» для доступу до неприватних полів суперкласу з підкласів, які маскують ці поля, оголошуючи власні поля з тими ж іменами.
- Для заборони розширення оголосіть метод або клас як final.
  - Якщо задати final void describe() для методу з класу Vehicle, компілятор повідомить про помилку при спробі переозначити цей метод у класі Car.
  - Ви не можете робити overriding method менш доступним, ніж метод, який переозначається.
    - Якщо метод describe() класу Car оголошено приватним, компілятор видасть помилку, оскільки приватний доступ більш закритий за пакетний (package) доступ (який за замовчуванням).
  - Проте describe() можна зробити більш доступним, оголошуючи його публічним: public void describe().

Переозначення

- Клас Car тепер має 2 методи describe(): явно оголошений та успадкований.
  - Метод void describe(String owner) не переозначує (override), а перевантажує (overload) метод describe() класу Vehicle.
  - Компілятор Java допомагає відстежити це, додаючи анотацію @Override

```
@Override
void describe()
{
   System.out.print("This car is a ");
   super.describe();
}
```

  - Використання анотації для переозначення є хорошою практикою та допомагає швидше знаходити помилки.

Наслідування реалізації

- Наслідування реалізації додає до порядку ініціалізації (лекція 2) декілька нових деталей:
  - Ініціалізатори суперкласу завжди виконуються до ініціалізаторів підкласів.
  - Конструктор підкласу завжди викликає конструктор суперкласу для ініціалізації object's superclass layer, а потім ініціалізує subclass layer.
- Підтримка реалізації наслідування в Java дозволяє лише розширювати один клас.
  - Ви не можете розширити декілька класів, оскільки це призведе до проблем.

Якби Java підтримувала множинне наслідування реалізації

```
class Bird
{
   void describe()
   {
      // code that outputs a description of a bird's appearance and behaviors
   }
}

class Horse
{
   void describe()
   {
      // code that outputs a description of a horse's appearance and behaviors
   }
}

public class FlyingHorse extends Bird, Horse
{
   public static void main(String[] args)
   {
      FlyingHorse pegasus = new FlyingHorse();
      pegasus.describe();
   }
}
```

- Структура класів показує проблему, коли як клас Bird, так і клас Horse оголошують свій метод describe().
- Який з цих методів повинен наслідувати FlyingHorse?
- Аналогічна проблема і з одноіменними полями, можливо навіть різних типів.

Ультимативний суперклас - Object

Метод	Опис
Object clone()	Створює та повертає копію поточного об’єкту.
boolean equals(Object obj)	Перевіряє поточний об’єкт на рівність obj.
void finalize()	Фіналізує поточний об’єкт.
Class<?> getClass()	Повертає об’єкт java.lang.Class для поточного об’єкта.
int hashCode()	Повертає хеш-код поточного об’єкта.
void notify()	«Будить» один з потоків, який очікує на монітор поточного об’єкта.
void notifyAll()	«Будить» усі потоки, які очікують на монітор поточного об’єкта.
String toString()	Повертає рядкове представлення поточного об’єкта.
void wait()	Змушує поточний потік очікувати на монітор поточного об’єкта, поки його не «розбудить» виклик notify() або notifyAll()
void wait(long timeout)	Змушує поточний потік очікувати на монітор поточного об’єкта, поки його не «розбудить» виклик notify() або notifyAll() чи не вийде заданий у мілісекундах час (timeout)
void wait(long timeout, int nanos)	Аналогічний метод, який уточнює час до наносекунд

Метод clone()

```
public class Employee implements Cloneable
{
   String name;
   int age;

   Employee(String name, int age)
   {
      this.name = name;
      this.age = age;
   }

   public static void main(String[] args) throws CloneNotSupportedException
   {
      Employee e1 = new Employee("John Doe", 46);
      Employee e2 = (Employee) e1.clone();
      System.out.println(e1 == e2); // Output: false
      System.out.println(e1.name == e2.name); // Output: true
   }
}
```

- Виконує дублювання об’єкту без виклику конструктора.
  - Виконує *поверхневе (shallow)* копіювання: значення полів одного об’єкта присвоюються відповідним полям клона.
- Метод main() використовує конструктор для ініціалізації нового об’єкта Employee та копіювання значень його полів (John Doe та 46) в об’єкт e2.
- Клас має реалізувати інтерфейс java.lang.Cloneable
  - Інакше екземпляри класу неможливо поверхнево клонувати за допомогою методу clone()

Демонстрація clone()

- Для перевірки відмінності об'єктів, посилання на які - e1 та e2, метод main() використовує оператор == та виводить результат типу Boolean – false.
- Для перевірки того, що об'єкт Employee було поверхнево клоновано, main() порівнює посилання на об'єкти типу Employee за допомогою оператора == та виводить булевий результат – true.
  - Метод clone() класу Object не переозначався, оскільки виклик e1.clone() трапився у класі, чиї екземпляри (instances) were to be cloned.
  - Для клонування об'єкту типу Employee з іншого класу (наприклад, UseEmployee) потрібно буде переозначити clone():

```
@Override
public Object clone() throws CloneNotSupportedException {
      return super.clone();
}
```

Глибоке (deep) клонування

- Поверхневе клонування не завжди бажане, оскільки первинний об’єкт та його клон звертаються (refer) до одного об’єкта по еквівалентних посилальних полях (reference fields).
  - У попередньому лістингу два об’єкти Employee звертались до одного об’єкта типу String по його полю name.
- Для класу String з незмінними об’єктами не проблема змінити mutable об’єкт за допомогою поля посилального типу від методу clone().
  - Проте це змушує початковий об’єкт бачити ці ж зміни за допомогою посилального поля.
  - Наприклад, додано посилкове поле hireDate до класу Employee.
  - Це поле типу Date та включає поля екземпляру year, month, day.
  - Оскільки клас Date змінний, можна змінити вміст цих полів у екземплярі Date, присвоєному hireDate.
- Припустимо, що планується змінювати дату клону, проте дату оригінального об’єкту типу Employee хочемо зберегти.
  - З поверхневим клонуванням це неможливо, оскільки зміни в копії видимі оригінальному об’єкту.
  - Потрібно змінити процедуру клонування так, щоб присвоювалось посилання new Date поля hireDate клону.
  - Така задача називається *глибоким копіюванням (клонуванням)*.

Глибоке клонування

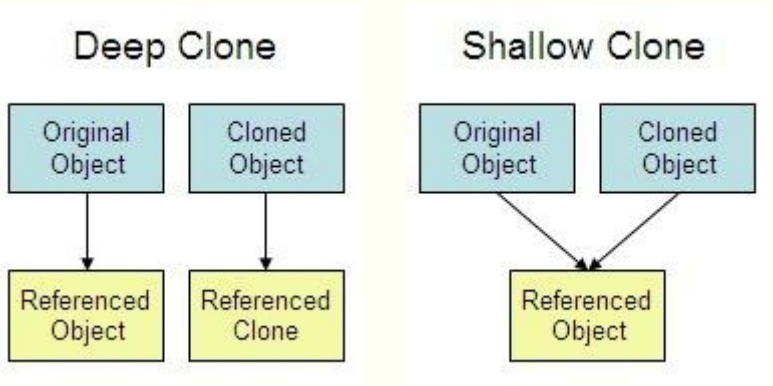
```
public class Employee implements Cloneable
{
   String name;
   int age;
   Date hireDate;
   Employee(String name, int age, Date hireDate)
   {
      this.name = name;
      this.age = age;
      this.hireDate = hireDate;
   }

   @Override
   protected Object clone() throws CloneNotSupportedException
   {
      Employee emp = (Employee) super.clone();
      if (hireDate != null) // no point cloning a null object (one that doesn't exist)
         emp.hireDate = new Date(hireDate.year, hireDate.month, hireDate.day);
      return emp;
   }

   public static void main(String[] args) throws CloneNotSupportedException
   {
      Employee e1 = new Employee("John Doe", 46, new Date(2000, 1, 20));
      Employee e2 = (Employee) e1.clone();
      System.out.println(e1 == e2); // Output: false
      System.out.println(e1.name == e2.name); // Output: true
      System.out.println(e1.hireDate == e2.hireDate); // Output: false
      System.out.println(e2.hireDate.year + " " + e2.hireDate.month + " " +
                         e2.hireDate.day); // Output: 2000 1 20
   }
}
```

```
class Date
{
   int year, month, day;

   Date(int year, int month, int day)
   {
      this.year = year;
      this.month = month;
      this.day = day;
   }
}
```



Рівність. Метод equals()

- Оператори == та != порівнюють два значення примітивного типу.
  - Також вони порівнюють два посилання, щоб перевірити, чи відносяться вони до одного об'єкта. Таке порівняння називають ***identity check***.
- Не можна використовувати == та != для логічного порівняння.
  - Два об'єкти Car з однаковими значеннями полів логічно еквівалентні, проте == повідомить про їх нерівність, оскільки посилання на них різні.
  - Оскільки == та != виконує найшвидше порівняння, клас String містить окрему підтримку порівняння literal strings та константних рядкових виразів за допомогою == та !=.
- Java постачає метод Object.equals().
  - Оскільки цей метод за замовчуванням порівнює посилання, його потрібно переозначувати (override) для порівняння вмісту об'єктів.
  - До переозначення equals() переконайтесь, що це необхідно.
  - Наприклад, клас java.lang.StringBuffer не переозначує equals().
  - Можливо, це задумка проектувальників.
- Ви не можете переозначити equals() довільним кодом.
  - Замість цього потрібно слідувати контракту, який прописано в документації Java для цього методу.

Властивості методу порівняння (умови контракту)

- Метод equals() реалізує відношення еквівалентності для не-null object references:
  - ***Рефлексивність***: для всіх не-null посилань x.equals(x) повертає true.
  - ***Симетричність***: для будь-якого не-null посилання на значення x та y, x.equals(y) повертає true, лише тоді, коли y.equals(x) повертає true.
  - ***Транзитивність***: для всіх не-null посилань на значення x, y та z, якщо x.equals(y) повертає true і y.equals(z) повертає true, то x.equals(z) теж повертає true.
  - ***Узгодженість*** (consistency): для будь-якого не-null посилання на значення x та y, багаторазові виклики x.equals(y) узгоджено повертають лише true або лише false, не надаючи інформації, що використовується для порівнянь в equals() on the objects is modified.
  - Для всіх не-null посилань на значення x, x.equals(null) повертає false.

Логічне порівняння об’єктів

```
public class Point
{
   private int x, y;

   Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   int getX()
   {
      return x;
   }

   int getY()
   {
      return y;
   }

   @Override
   public boolean equals(Object o)
   {
      if (!(o instanceof Point))
         return false;
      Point p = (Point) o;
      return p.x == x && p.y == y;
   }
```

- Переозначення методу equals() починається з умови перевірки того, чи є переданий параметр о екземпляром класу Point.
  - Якщо ні, відразу повертаємо false.

```
   public static void main(String[] args)
   {
      Point p1 = new Point(10, 20);
      Point p2 = new Point(20, 30);
      Point p3 = new Point(10, 20);
      // Test reflexivity
      System.out.println(p1.equals(p1)); // Output: true
      // Test symmetry
      System.out.println(p1.equals(p2)); // Output: false
      System.out.println(p2.equals(p1)); // Output: false
      // Test transitivity
      System.out.println(p2.equals(p3)); // Output: false
      System.out.println(p1.equals(p3)); // Output: true
      // Test nullability
      System.out.println(p1.equals(null)); // Output: false
      // Extra test to further prove the instanceof operator's usefulness.
      System.out.println(p1.equals("abc")); // Output: false
   }
}
```

Виконання контракту

- Вираз (o instanceof Point) задовольняє останню умову контракту: x.equals(null) повертає false.
  - Оскільки null-посилання не є екземпляром у жодному класі, передача цього значення в метод equals() завжди повертає false.
- Також цей вираз не дає відбутись викиду екземпляру виключення java.lang.ClassCastException через вираз зведення типу (Point) o тоді, коли передається об'єкт типу, відмінного від Point у метод equals().
  - Після зведення типу вимоги контракту (крім узгодженості) виконуються лише дозволяючи порівняння одних Point-об'єктів з іншими за допомогою виразу p.x == x && p.y == y.
  - Узгодженість виконується за умови детермінованості методу equals() – метод не використовує поля, значення яких можуть змінюватись від виклику до виклику.
- Важливо завжди переозначати метод hashCode() при переозначенні методу equals().

Хеш-коди. Метод hashCode()

- Метод повертає 32-бітне ціле число, яке ідентифікує хеш-код об’єкта.
  - Протягом виконання Java-додатку виклик hashCode() для одного і того ж об’єкта має повертати однакове ціле число.
    - не вказуючи використану в порівняннях методом equals(Object) інформацію про об’єкт, який модифікується.
  - Якщо два об’єкта рівні згідно з методом equals(Object), тоді виклик методу hashCode() для цих об’єктів має повертати однаковий результат.
  - Якщо два об’єкта нерівні згідно з методом equals(Object), тоді виклик методу hashCode() для цих об’єктів не обов’язково має повертати різний integer результат.
    - Проте програміст повинен знати, що отримування різних результатів для нерівних об’єктів може покращити продуктивність хеш-таблиць.

Порушення контракту

- Якщо переозначити equals(), але не hashCode(), порушите другий пункт контракту: хеш-коди однакових об'єктів повинні бути однаковими.
- Серйозні наслідки:

```
java.util.Map<Point, String> map = new java.util.HashMap<Point, String>();
map.put(p1, "first point");
System.out.println(map.get(p1)); // Output: first point
System.out.println(map.get(new Point(10, 20))); // Output: null
```

- Припустимо, цей лістинг додається в метод main() зі слайду 55.
  - Префікс java.util. та <Point, String> відносяться до пакетів та дженериків (наступна лекція).

Представлення рядків

- Метод toString() повертає рядкове представлення поточного об’єкта.
  - Якщо вивести об’єкт, буде показано запис у формі
  - *клас-об’єкта@хеш-код-об’єкта*
  - Наприклад, Point@3e25a5
- Краще переозначити toString(), щоб метод повертав змістовний опис об’єкта.
  - Наприклад,

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

- Цього разу після виклику System.out.println(p1); виведе (10, 20).

Фіналізація. Метод finalize()

- Відноситься до збірки сміття.
  - finalize() викликається збирачем сміття до об'єкту, коли на об'єкт більше немає посилань.
  - Підклас переозначає (override) метод finalize(), щоб вивільнити системні ресурси або виконати іншу очистку.
- Версія методу для класу Object нічого не робить – необхідно переозначувати цей метод для будь-якого потрібного cleanup code.
  - Оскільки ВМ може ніколи не викликати finalize() до завершення роботи додатку, краще задавати явний метод для очистки.
- Ніколи не покладайтесь на finalize() для вивільнення обмежених ресурсів, наприклад, дескрипторів файлів.
  - Наприклад, якщо об'єкт з додатку відкриває файли, очікуючи, що його метод finalize() закриє їх, існує можливість, що додаток не зможе відкрити додаткові файли, коли віртуальна машина повільна.
  - Ще гірше те, що finalize() може викликатись на іншій віртуальній машині, отримуючи проблему too-many-open-files, яка про себе ніяк не сповіщає.
  - Розробник може хибно вважати, що додаток поводиться узгоджено на різних віртуальних машинах.

Переозначення finalize()

- Рівень підкласу об’єкту повинен дати рівню суперкласу можливість виконати фіналізацію.
  - Виконати можна, викликавши super.finalize();

```
@Override
protected void finalize() throws Throwable
{
   try
   {
      // Perform subclass cleanup.
   }
   finally
   {
      super.finalize();
   }
}
```

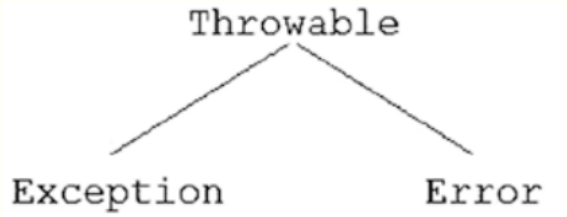
Якщо викинуто виняток, виходимо з методу.
За відсутності try-finally виклик super.finalize(); ніколи не виконується.

Інакше логіка обробки винятків Java виконує метод finalize() із суперкласу.

Можливість воскресіння (resurrection)

- Метод finalize() часто використовується для воскресіння об’єкту: перетворення об’єкта без посилань на нього в об’єкт з посиланнями.
  - для реалізації пулів об’єктів, які видаляють витратні (time-wise) для створення, однакові об’єкти.
  - Яскравий приклад – об’єкти підключення до БД.
- Воскресіння відбувається, коли Ви присвоюєте посилання на поточний об’єкт іншій довгоживучій змінній.
  - Наприклад, можна задати у finalize(): r = this;
- Через можливість воскресіння існує серйозне зниження продуктивності, накладене на збірку сміття об’єкта, який переозначує finalize().
  - Фіналізатор воскреслого об’єкта неможливо викликати знову.

Ієрархія класів Throwable



- Java постачає ієрархію класів, які представляють різні види виключень.
  - Корінь – java.lang.Throwable, суперклас для всіх *throwables* (виключень та error-об'єктів).
  - Exception – кореневий клас для всіх exception-oriented throwables.
  - Error – кореневий клас для всіх error-oriented throwables.
- Публічні методи класу часто викликають допоміжні методи, що обробляють різні винятки.
  - Публічний метод може ***обгорнути низькорівневий виняток у високорівневому винятку***, який задокументовано в інтерфейсі публічного методу.
- Обгорнутий виняток називають *cause,* оскільки його існування спричиняє (causes) викид високорівневого винятку.
  - cause створюється викликом конструктору Throwable(Throwable cause) або Throwable(String message, Throwable cause), який викликає метод initCause(), щоб зберігати cause.
  - Альтернатива: напряму викликати initCause(), проте це слід робити негайно після створення throwable.
  - Щоб повернути cause, викликають метод getCause().
- Коли викинуто виняток, за ним залишається стек незавершених викликів методів.
  - Конструктори Throwable викликають метод fillInStackTrace(), щоб записати трасувальну інформацію, яку можна вивести, викликавши printStackTrace().

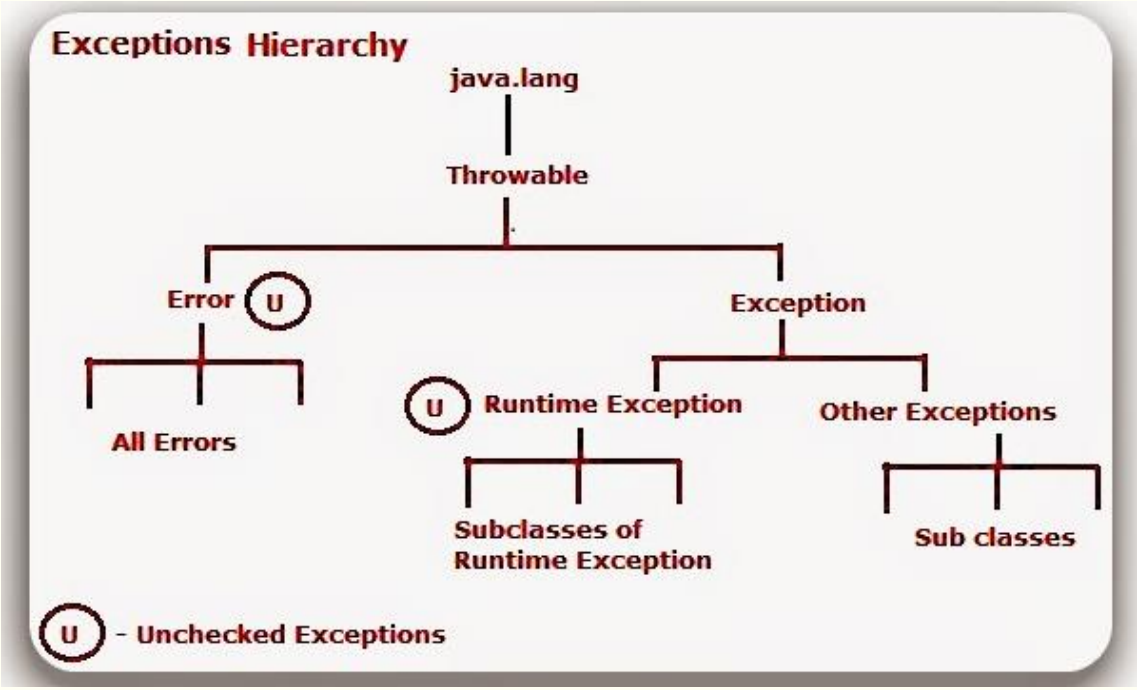
Method	Description
`Throwable()`	Create a throwable with a null detail message and cause.
`Throwable(String message)`	Create a throwable with the specified detail message and a null cause.
`Throwable(String message, Throwable cause)`	Create a throwable with the specified detail message and cause.
`Throwable(Throwable cause)`	Create a throwable whose detail message is the string representation of a nonnull cause or null.
`Throwable fillInStackTrace()`	Fill in the execution stack trace. This method records information about the current state of the stack frames for the current thread within this throwable. (I discuss threads in Chapter 7.)
`Throwable getCause()`	Return the cause of this throwable. When there is no cause, null is returned.
`String getMessage()`	Return this throwable's detail message, which might be null.
`StackTraceElement[] getStackTrace()`	Provide programmatic access to the stack trace information printed by `printStackTrace()` as an array of stack trace elements, each representing one stack frame.
`Throwable initCause(Throwable cause)`	Initialize the cause of this throwable to the specified value.
`void printStackTrace()`	Print this throwable and its backtrace of stack frames to the standard error stream.
`void setStackTrace(StackTraceElement[] stackTrace)`	Set the stack trace elements that will be returned by `getStackTrace()` and printed by `printStackTrace()` and related methods.

- Метод getStackTrace() забезпечує програмний доступ до stack trace, повертаючи цю інформацію в масиві екземплярів типу java.lang.StackTraceElement—кожен екземпляр представляє одне входження (entry).
- Клас StackTraceElement постачає методи для повернення stack trace інформації.
  - Наприклад, String getMethodName() повертає назву незавершеного методу.
  - Метод setStackTrace() розроблено для використання фреймворками Remote Procedure Call (RPC) та іншими advanced systems, дозволяючи клієнту переозначати stack trace за замовчуванням, згенерований методом fillInStackTrace(), коли throwable конструюється або десеріалізується при його зчитуванні з потоку серіалізованих даних (serialization stream).
- Класи java.lang.Exception та java.lang.Error пропонують по 4 конструктора, які передають свої аргументи у відповідники з Throwable, проте не постачають інших методів, крім успадкованих від Throwable.
  - Клас Exception сам субкласується java.lang.CloneNotSupportedException, java.lang.IOException та іншими класами.
  - Аналогічно, клас Error субкласується java.lang.AssertionError, java.lang.OutOfMemoryError та іншими класами.

Checked Exceptions vs. Runtime Exceptions

- ***Виняток, що перевіряється*** (*checked exception)* – це виняток, який представляє проблему з можливістю відновлення роботи, для якого розробник має забезпечити обхідний шлях (workaround).
  - Компілятор перевіряє код, щоб переконатись, що виключення оброблено в методі, де воно викидається, або явно позначається обробленим десь в іншому місці.

- Клас Exception та всі підкласи, крім java.lang.RuntimeException та його підкласів, описують виключення, що перевіряються.
  - Наприклад, класи CloneNotSupportedException та IOException описують checked exceptions.
  - CloneNotSupportedException має не перевірятись, оскільки немає runtime workaround для цього виду виключення.



Винятки, що не перевіряються

- ***Винятки часу виконання*** (*runtime exception)* – це винятки, що представляють помилку в коді.
  - Також називаються ***винятками, що не перевіряються*** (*unchecked exception),* оскільки його не потрібно обробляти або явно ідентифікувати—помилка має виправлятись.
  - Оскільки такі винятки можуть викидатись в багатьох місцях, їх було б важко обробляти.
- Клас RuntimeException та його підкласи описують unchecked-винятки.
  - java.lang.ArithmeticException описує арифметичні проблеми, наприклад, ділення на 0.
  - java.lang.ArrayIndexOutOfBoundsException викидається при спробі отримати доступ до елементу масиву з від'ємним індексом або з індексом, більшим за довжину масиву.

Власні Exception-класи

- Визначтесь, чи немає потрібного відповідника серед стандартних класів.
  - Якщо немає, виберіть, який клас субкласувати: Exception чи RuntimeException.
  - Рекомендується субкласувати RuntimeException, якщо Ваш клас буде описувати coding mistake.
- Рекомендується називати свій клас з суфіксом Exception.
  - Припустимо, створюється клас Media, чиї статичні методи виконують мультимедійні задачі.
  - Наприклад, один метод конвертує аудіофайли не-MP3 формату в MP3.
  - Методу в якості аргументів передаватимуться початковий та результуючий файли.
  - Перед конвертацією метод має переконатись, що формат вхідного файлу узгоджується з форматом, записаним у його розширенні. Якщо це не так, викидається виняток.
  - Також виняток повинен зберігати очікуваний та наявний медіа-формати, щоб обробник їх визначив при виводі повідомлення для користувача.
  - Оскільки доречного Java-класу для такого винятку немає, введемо клас InvalidMediaFormatException.

Оголошення власного Exception-класу

```
package media;

public class InvalidMediaFormatException extends Exception
{
   private String expectedFormat;
   private String existingFormat;
   public InvalidMediaFormatException(String expectedFormat,
                                      String existingFormat)
   {
      super("Expected format: " + expectedFormat + ", Existing format: " +
            existingFormat);
      this.expectedFormat = expectedFormat;
      this.existingFormat = existingFormat;
   }

   public String getExpectedFormat()
   {
      return expectedFormat;
   }

   public String getExistingFormat()
   {
      return existingFormat;
   }
}
```

- InvalidMediaFormatException постачає конструктор, що викликає конструктор public Exception(String message) класу Exception з детальним повідомленням, яке включає очікуваний та наявний формати.

Корисно зберігати такі деталі в повідомленні, оскільки можуть бути проблеми з відтворенням винятку.

- InvalidMediaFormatException також постачає методи getExpectedFormat() та getExistingFormat(), що повертають ці формати.

Це повідомлення, можливо, потрібно буде локалізувати іншими мовами.

Викидання винятків

```
package media;

import java.io.IOException;

public final class Media
{
   public static void convert(String srcName, String dstName)
      throws InvalidMediaFormatException, IOException
   {
      if (srcName == null)
         throw new NullPointerException(srcName + " is null");
      if (dstName == null)
         throw new NullPointerException(dstName + " is null");
      // Code to access source file and verify that its format matches the
      // format implied by its file extension.
      //
      // Assume that the source file's extension is RM (for Real Media) and
      // that the file's internal signature suggests that its format is
      // Microsoft WAVE.
      String expectedFormat = "RM";
      String existingFormat = "WAVE";
      throw new InvalidMediaFormatException(expectedFormat, existingFormat);
   }
}
```

- Оголошується константний клас Media (utility-клас, що містить лише методи класу, і не буде розширюватись).
- Оператор *throws* у методі Media.convert() ідентифікує всі checked-винятки, що викидаються за межі методу та мають оброблятись іншим методом.
  - Він вказує на те, що метод здатний викидати екземпляри InvalidMediaException або IOException віртуальній машині.
  - Також convert() демонструє оператор throw, який викидає екземпляр Throwable або його підкласу.
  - Оператор викидає цей екземпляр віртуальній машині, яка потім шукає підходящий обробник для виключення.

NullPointerException

- Вид винятку, що викидається за умови, що аргумент некоректний (invalid).
  - Клас java.lang.IllegalArgumentException узагальнює illegal argument scenario to include other kinds of illegal arguments.
  - Наприклад, даний метод викидає об'єкт IllegalArgumentException, коли числовий аргумент буде від'ємним.

```
public static double sqrt(double x)
{
   if (x < 0)
      throw new IllegalArgumentException(x + " is negative");
   // Calculate the square root of x.
}
```

Спрощення обробки помилок

- Потрібно easily manage the closing of effectively final variables.
  - Оператор try-with-resources з'явився в Java 7 та спрощує управління ресурсами.
  - Ресурс – це об'єкт, який необхідно закрити, як тільки програма перестає його використовувати.
  - Наприклад, File-ресурси, JDBC-ресурси, Socket-ресурси тощо.
- До Java 7 не було автоматичного управління ресурсами, зазвичай всі операції записувались у блоці finally.
  - Такий підхід вів до витоків пам'яті та падіння продуктивності, якщо забули закрити ресурс.
  - У Java 9 стало ще простіше: можна не створювати нову змінну в конструкції (приклад з writeFile()):

```
public static void main(String[] args) {
    try {
        writeFile(new BufferedWriter(
                new FileWriter("Easy TryWithResources")),
                "This is easy in Java 9");
    } catch (IOException ioe) {
        System.out.println(ioe);
    }
}
```

Порівняння реалізацій до Java 9 та після

До Java 9

```
public static void writeFile(BufferedWriter writer, String text) {
   try (BufferedWriter w = writer) {
        w.write(text);
    } catch (IOException ioe) {
        System.out.println(ioe);
    }
}
```

Java 9+

```
public static void writeFile(BufferedWriter writer, String text) {
    try (writer) {
        writer.write(text);
    } catch (IOException ioe) {
        System.out.println(ioe);
    }
}
```

ДЯКУЮ ЗА УВАГУ!

Наступне запитання: інкапсуляція та приховування інформації в Java