

A decorative graphic on the left side of the slide, consisting of a network of thin, light green lines and small circles, resembling a circuit board or a stylized tree structure.

JAVA ООП. СПАДКУВАННЯ ТА
ПОЛІМОРФІЗМ. ІНТЕРФЕЙСИ.
ФАЙЛОВІ ПОТОКИ ТА ОПРАЦЮВАННЯ
ВИНЯТКІВ.

PACKAGES

- **Java package** is a mechanism for organizing Java classes into **namespaces**
- Programmers also typically use packages to organize classes belonging to the same category or providing similar functionality.
 - A package provides a **unique** namespace for the types it contains.
 - Classes in the same package can access each other's package-access members.

```
package java.awt.event;
```

PACKAGING

java.lang – basic language functionality and fundamental types

java.util – collection data structure classes

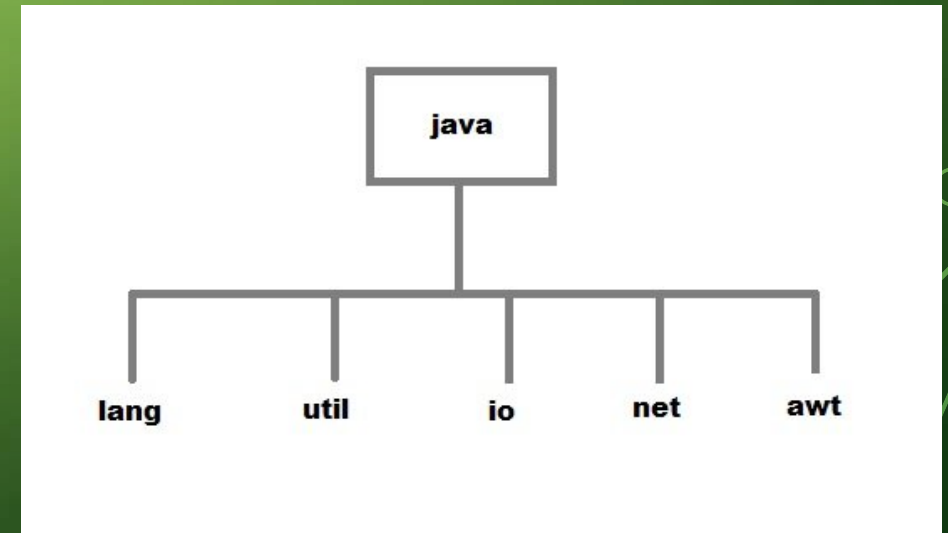
java.io – file operations

java.math – multiprecision arithmetics

java.awt – basic hierarchy of packages for native GUI components

javax.swing – hierarchy of packages for platform-independent rich GUI components

The **java.lang.*** package is available **without** the use of an import statement.



ІМПОРТ ПАКЕТІВ

Імпорт пакетів

Якщо класи знаходяться в іншому пакеті, їх потрібно імпортувати. вказувати повну назву пакету перед використанням класом:

```
java.util.Date today = new java.util.Date();
```

Такий спосіб незручний, краще використати інструкцію `import`:

```
import java.util.Date; // імпортуємо клас
```

Далі можна використовувати:

```
Date today = new java.util.Date();
```

СТАТИЧНИЙ ІМПОРТ

Починаючи з Java SE 5.0 можна зробити так:
`import static java.lang.System.*;`

Далі використовуємо:

```
out.println("Goodbye, World!"); // i.e., System.out  
exit(0); // i.e., System.exit
```

JAVA OOPS CONCEPTS

- **Encapsulation** Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines. A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.
- **Inheritance** When one object acquires all the properties and behaviors of parent object i.e., known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.
- **Polymorphism** When one task is performed by different ways i.e., known as polymorphism. For example: cat speaks meow, dog barks woof etc.

INHERITANCE VS. COMPOSITION

```
class Point {  
    protected int x, y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
class Circle extends Point {  
    private int radius;  
    public Circle(int x, int y, int radius) {  
        this.x = x;  
        this.y = y;  
        this.radius = radius;  
    }  
}
```

```
Circle circle = new Circle(74, 38, 26);
```

COMPOSITION

- **Composition** is the design technique to implement **has-a** relationship in classes.
- Composition is achieved by using *instance variables* that *refers* to other objects.

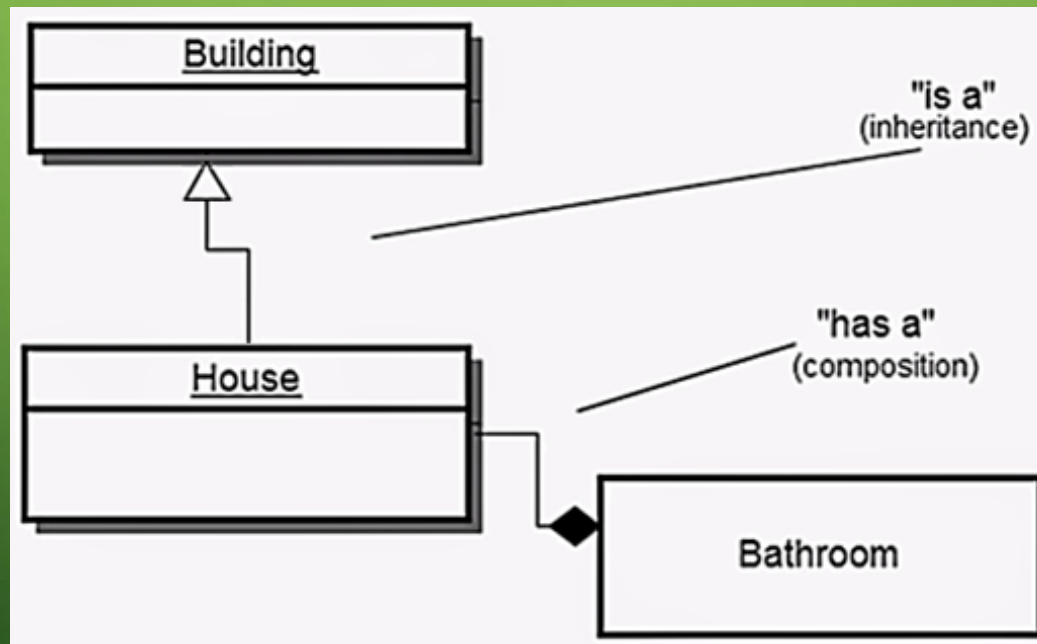
```
class Point {  
    private int x, y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
class Circle {  
    private Point point;  
    private int radius;  
    public Circle(Point point, int radius) {  
        this.point = point;  
        this.radius = radius;  
    }  
}
```

```
Point p = new Point(74, 38);  
Circle circle = new Circle(p, 26);
```

INHERITANCE VS. COMPOSITION

- Kinds of **Relationships** between objects:
 - **"is a"** - object of subclass **"is a"** object of the superclass (**inheritance**).
 - **"has a"** – object **"has a"** object of another class as a member (**composition**)



ІНТЕРФЕЙС

Інтерфейс - це елемент мови програмування який, специфікує набір послуг, що надаються класом. Іншими словами інтерфейс - це елемент мови програмування який містить опис методів, але не містить їх реалізацію. Оголошення інтерфейсів дуже схоже на спрощене оголошення класів. Воно починається з заголовка. Спочатку вказуються модифікатори. Інтерфейс може бути оголошений як `public` і тоді він буде доступний для загального використання, або модифікатор доступу може не вказуватися, в цьому випадку інтерфейс доступний тільки для типів свого пакета.

Interfaces

```
public interface Worker {  
    int getSalary(); // public abstract  
}  
  
public class Director implements Worker {  
    public int getSalary() {  
        // method definition here  
        return 0;  
    }  
}
```

Interfaces

```
interface Volumetric {  
    double PI = 3.14;  
  
    double getVolume();  
  
    static double getPI() {  
        return Volumetric.PI;  
    }  
    default String info() {  
        return definition() +  
            "1 litre = (10 cm)^3 = 1000 cubic centimetres = 0.001 cubic metres";  
    }  
    private String definition() {  
        return "Volume is the quantity of three-dimensional space" +  
            "enclosed by a closed surface.\n";  
    }  
}
```

Interfaces

- To access the interface methods, the interface must be "implemented by another class with the **implements** keyword.
- Example:

```
public class Ball extends Shape implements Volumetric {  
    private double radius;  
    public Ball(double radius, String name) {  
        super(name); this.radius = radius;  
    }  
    @Override  
    public double getArea() {  
        return 4 * Math.PI * radius * radius;  
    }  
    @Override  
    public double getVolume() {  
        return 4.0 / 3 * Volumetric.PI * Math.pow(radius, 3);  
    }  
}
```

Multiple interfaces implementation

- To implement *multiple interfaces*, separate them with a comma:

```
class Cube extends Shape implements Vertexable, Volumetric {  
    private double side;  
    public Cube(double side, String name) {  
        super(name);  
        this.side = side;  
    }  
    @Override  
    public double getArea() {  
        return 12 * side;  
    }  
    @Override  
    public double getVolume() {  
        return Math.pow(side, 3);  
    }  
    @Override  
    public int getNumberOfVertex() {  
        return 8;  
    }  
}
```

```
interface Vertexable {  
    int getNumberOfVertex();  
}  
  
interface Volumetric {  
    double getVolume();  
}
```

Interfaces with same method

- If a type implements **two interfaces**, and each interface define a method that has **identical signature**, then in effect there is only one method, and they are **not distinguishable**.

```
public interface Vertexable {  
    ...  
    void info();  
}
```

```
public interface Volumetric {  
    ...  
    void info();  
}
```

```
public class Cube extends Shape implements Vertexable, Volumetric {  
    ...  
    @Override  
    public void info() { ... }  
}
```

Note that there is **only one** `@Override` **necessary**. This is because `Vertexable.info` and `Volumetric.info` are "Override-equivalent"

INTERFACE COMPARABLE

- Interface `Comparable` allows *custom sorting* of objects when implemented.
 - When a class implements this interface, we must add the public method `compareTo(Object o)`.

```
public class Person implements Comparable<Person> {  
    private String name;  
    private int age;  
    // constructors, getters, setters, toString  
  
    @Override  
    public int compareTo(Person p) {  
        if (this.name.compareTo(p.name) != 0 )  
            return this.name.compareTo(p.name);  
        else  
            return Integer.compare(this.age, p.age);  
    }  
}
```

INTERFACE COMPARATOR

Interface Comparator allows *custom sorting* of objects when implemented.

When a class implements this interface, we must add the public method `compare (Object o1, Object o2)`.

Methods compare can throw an exception *ClassCastException*, if the object types are not compatible in the comparison.



Застосування інтерфейсів

Реалізація інтерфейсів не обов'язково повинна здійснюватися в окремому класі. Допускається реалізація абстрактних методів інтерфесу безпосередньо у місці створення об'єкта (локальної змінної, поля класу чи параметра методу). В такому випадку говорять про створення анонімного типу(класу) - віртуальна машина створює новий тип, даючи йому системну назву, але створювати інші об'єкти цього ж типу не вдасться.

Особливе місце займають інтерфейси, що мають оголошений рівно один метод. Їх називають функціональними інтерфейсами і їх можна використовувати для передачі методу в якості параметра іншого методу, на кшталт вказівників на функцію у C та C++.

Лямбда-вирази

Анонімна реалізація функціональних інтерфейсів починаючи з Java 8 отримала новий синтаксис у формі популярних в інших мовах лямбда-виразів. Він значно простіший ніж у C++, бо відсутній оператор захоплення змінних оточення (всі вони захоплюються автоматично за принципом вбудованого класу), а тип результату виводиться з типу об'єкта, в який перетворюється лямбда-вираз.

Тобто лямбда складається з круглих дужок, знака лямбда-виразу «->» та блоку коду.



Лямбда-вирази

Анонімна реалізація функціональних інтерфейсів починаючи з Java 8 отримала новий синтаксис у формі популярних в інших мовах лямбда-виразів. Він значно простіший ніж у C++, бо відсутній оператор захоплення змінних оточення (всі вони захоплюються автоматично за принципом вбудованого класу), а тип результату виводиться з типу об'єкта, в який перетворюється лямбда-вираз.

Тобто лямбда складається з круглих дужок, знака лямбда-виразу «->» та блоку коду.

Лямбда-вирази

Вбудовані функціональні інтерфейси:

```
Predicate<T>    {boolean test(T t);}
```

```
Consumer<T>{void accept(T t);}
```

```
Function<T,R>{R apply(T t);}
```

```
Supplier<T>    {T get();}
```

```
UnaryOperator<T> {T apply(T t);}
```

```
BinaryOperator<T> {T apply(T t1, T t2);}
```

ПОТОКИ ВВОДУ-ВИВОДУ

	Байтові	Символьні
Базові класи потоків	InputStream	Reader
	OutputStream	Writer
Фільтровані потоки	FilterInputStream	FilterReader
	FilterOutputStream	FilterWriter
Буферизовані потоки	BufferedInputStream	BufferedReader
	BufferedOutputStream	BufferedWriter
Канальні потоки	PipedInputStream	PipedReader
	PipedOutputStream	PipedWriter
Потоки для роботи з файлами	FileInputStream	FileReader
	FileOutputStream	FileWriter

КЛАС FILE

Конструктори

`File(File parent,
String child)`

`File(String pathname)`

`File(String parent,
String child)`

`File(URI uri)`

Деякі методи

`getName()`

`getAbsolutePath()`

`exists()`

`isDirectory()`

`length()`

`renameTo(File file)`

`createNewFile()`

ВИБІР ФАЙЛА

```
JFileChooser jFileChooser = new JFileChooser();  
int result = jFileChooser.showOpenDialog(null);  
File file;  
if(result==JFileChooser.APPROVE_OPTION){  
    file = jFileChooser.getSelectedFile();  
    System.out.println("File: " + file.getName());  
    try {  
        ..  
    }  
}
```

ОБРОБКА ВИНЯТКОВИХ СИТУАЦІЙ

При виникненні виключення в блоці `try` управління переходить в блок `catch`, який може обробити цей виняток. Якщо такого блоку не знайдено, то користувачеві відображається повідомлення про необробленому виключення, а подальше виконання програми зупиняється. І щоб такої зупинки не відбулося, і треба використовувати блок `try..catch`.

ОБРОБКА ВИНЯТКОВИХ СИТУАЦІЙ

```
try{  
    int[] numbers = new int[3];  
    numbers[4]=45;  
    System.out.println(numbers[4]);  
}  
catch(Exception ex){  
    ex.printStackTrace();  
}  
System.out.println("Програма завершена");
```

КЛАСИ ВИНЯТКІВ

У Java є багато різних типів винятків, і ми можемо розмежувати їх обробку, включивши додаткові блоки catch:

```
int[] numbers = new int[3];  
try{    numbers[6]=45;    numbers[6]=Integer.parseInt("gfd");  
} catch (ArrayIndexOutOfBoundsException ex) {  
    System.out.println("Выход за пределы массива");  
} catch(NumberFormatException ex){  
    System.out.println("Помилка перетворення в число");}
```

ОПЕРАТОР THROW

Щоб повідомити про виникнення виняткової ситуації в програмі, можна використовувати оператор **throw**. Тобто за допомогою цього оператора ми самі можемо створити виняток і викликати його в процесі виконання. Наприклад

```
if(x>=30){ throw new Exception("Число x завелике");}
```

У блоці `catch` ми можемо отримати повідомлення про виключення з допомогою методу `getMessage ()`.

ОПЕРАТОР THROWS

Іноді метод, в якому може генеруватися виняток, сам не обробляє це виняток. В цьому випадку в оголошенні методу використовується оператор `throws`, який треба обробити при виклику цього методу. Наприклад, у нас є метод обчислення факторіала, і нам треба обробити ситуацію, якщо в метод передається число менше 1:

```
public static int getFactorial(int num) throws Exception{  
    }  
}
```

КЛАСИ ВИНЯТКІВ

Базовим класом для всіх винятків є клас **Throwable**. Від нього вже успадковуються два класи: **Error** і **Exception**. Всі інші класи є похідними від цих двох класів.

Клас **Error** описує внутрішні помилки в виконуючій середовищі Java. Програміст має дуже обмежені можливості для обробки подібних помилок.

Власне виключення успадковуються від класу **Exception**. Серед цих винятків слід виділити клас **RuntimeException**. **RuntimeException** є базовим класом для так званої групи **unchecked**-винятків. Компілятор не перевіряє факт обробки таких винятків і їх можна не вказувати разом з оператором **throws** в оголошенні методу.

КЛАСИ ВИНЯТКІВ

Деякі unchecked-винятки:

- **ArithmeticException**: виняток ділення на нуль
- **IndexOutOfBoundsException**: індекс вийшов за межі масиву
- **IllegalArgumentException**: неправильні аргументи при виклаку методу
- **NullPointerException**: використання порожнього посилання на об'єкт
- **NumberFormatException**: помилка перетаврення стрічки в число

Інші нащадки класу Exception, належать до checked-винятків.