

GENERICCS

Розглянемо приклад:

```
import java.util.ArrayList;
import java.util.List;
public class Test {
    public static void main(String[] args) {
        List arr = new ArrayList();
        String o = "Hello";
        arr.add(o);
        String i = arr.get(0); // помилка
        System.out.println(i);
    }
}
```

Результат:

Error:(8, 22) java: incompatible types: java.lang.Object cannot be converted to java.lang.String.

Метод add(E e) дозволяє використати як параметр елемент довільного типу (для якого батьківським є тип [Object](#)). Того ж типу ([Object](#)) буде і результат, котрий повертає метод get(). Тобто, додавання до ArrayList елемента типу String не означає, що буде повернуто елемент типу String. Натомість отримаємо посилання на елемент батьківського супертипу Object.

Приведемо тип елемента, котрий повертає метод get(), до потрібного нам String, скориставшись cast:

```
import java.util.ArrayList;
import java.util.List;
public class Test {
    public static void main(String[] args) {
        List arr = new ArrayList();
        String o = "Hello";
        arr.add(o);
        String i = (String)arr.get(0); // перетворення типу
        System.out.println(i);
    }
}
```

Результат:

Hello

Неважко помітити, що такий спосіб додавання елементів до ArrayList та отримання їх несе потенційну небезпеку виникнення помилки виконання, оскільки ніде не контролюється відповідність типів на цих двох етапах.

Змінимо тип параметра, котрий додаємо до ArrayList, на int. Зверніть увагу на те, що компілятор не зустрічає жодних помилок і код успішно компілюється. Проте, помилка виникає під час виконання і пов'язана вона із перетворенням String(Integer):

```
import java.util.ArrayList;
import java.util.List;
public class Test {
    public static void main(String[] args) {
        List arr = new ArrayList();
        int o = 1;
        arr.add(o);
        String i = (String)arr.get(0); // помилка
        System.out.println(i);
    }
}
```

Результат:

Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

Знаходження причин помилок виконання займає набагато більше часу, ніж виявлення та усунення їх на етапі компіляції.

Generics дозволяє типам (класам та інтерфейсам) бути параметрами при оголошенні класів, інтерфейсів та методів. Подібно до формальних параметрів, що використовують при оголошенні методів, параметри типу дозволяють повторно використовувати код для різних вхідних значень. Різниця у тому, що вхідні величини для локальних параметрів — це значення, а для параметрів типу — типи.

Код, що використовує [generics](#):

- надає строгу перевірку типів при компіляції;
- дозволяє уникнути cast;
- надає можливість реалізувати узагальнені (generic) алгоритми, що працюють з колекціями різних типів, можуть бути кастомізовані, безпечніші щодо використання типів та легші для читання.

Перепишемо наш приклад з використанням generics:

```
import java.util.ArrayList;
import java.util.List;
public class Test {
    public static void main(String[] args) {
        List<String> arr = new ArrayList<>();
        String o = "Hello";
        arr.add(o);
        String i = arr.get(0); // перетворення типу
        System.out.println(i);
    }
}
```

Результат:

Hello

Узагальнені (generic) типи

Джерело

Узагальненим типом називають узагальнений клас або інтерфейс, параметрами якого є типи. Наведений нижче приклад демонструє суть концепції.

Розглянемо клас Box, що оперує об'єктами довільного типу.

```
class Box {  
    private Object object;  
    public void set(Object object) {  
        this.object = object;  
    }  
    public Object get() {  
        return object;  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Box myBox = new Box();  
        myBox.set("Hello");  
        System.out.println(myBox.get());  
        myBox.set(5);  
        System.out.println(myBox.get());  
        myBox.set(5.0);  
        System.out.println(myBox.get());  
    }  
}
```

Результат:

Hello

5

5.0

Оскільки методи set() та get() оперують типом Object, параметри можуть бути довільними, проте не (int, float, double тощо) примітивними. У випадкові примітивних типів використовують класи-оболонки (детальніше: [тут](#)). Аутопакування та розпакування відбувається автоматично.

Під час компіляції неможливо встановити, як клас буде використано під час роботи програми. Одна частина програми може помістити в об'єкт Integer у той час, як інша — намагатиметься отримати з нього String. У результаті трапиться помилка виконання.

Напишемо узагальнену (generic) версію цього класу.

Generic клас оголошують наступним чином:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

Секція, що знаходиться після імені класу і обмежена знаками < та >, визначає параметр типу (також їх ще називають змінними-типами). Змінні T1, T2, ..., Tn — це змінні-типи.

Таким чином, generic-версія класу Box матиме вигляд:

```
class Box<T> {  
    private T object;  
    public void set(T object) {  
        this.object = object;  
    }  
    public T get() {  
        return object;  
    }  
}  
public class Test {  
    public static void main(String[] args) {  
        Box<String> myBox1 = new Box<>();  
        Box<Integer> myBox2 = new Box<>();  
        Box<Double> myBox3 = new Box<>();  
        myBox1.set("Hello");  
        myBox2.set(5);  
        myBox3.set(5.0);  
        System.out.println(myBox1.get());  
        System.out.println(myBox2.get());  
        System.out.println(myBox3.get());  
    }  
}
```

Результат:

```
Hello  
5  
5.0
```

Спроба ініціалізувати поле об'єкта (myBox1, myBox2 чи myBox3) значенням невласного йому типу викличе помилку компіляції.

Змінна-тип може бути довільним непримітивним типом: класом, інтерфейсом, масивом чи іншою змінною-типом.

Таким самим підхід застосовують до узагальнених інтерфейсів.

Угода про імена параметрів типу

За угодою імена параметрів типу — це одинарні літери верхнього регістру. Завдяки угоді імена типів відрізняються від [імен змінних](#).

Найбільш вживані імена параметрів-типів наступні:

- E - Element (використовується у Java Collections Framework)
- K – Key (Ключ);
- N – Number (Число);
- T - Type (Тип);
- V - Value (Значення);
- S,U,V etc. - 2й, 3й, 4й типи.

Виклик та створення екземпляру узагальненого типу

Посилання на узагальнений клас (у нашому випадкові — Box) у тілі програми реалізують, попередньо здійснивши виклик узагальненого типу із заміною узагальненого параметра T на конкретне значення типу:

```
Box<String> myBox1;  
Box<Integer> myBox2;  
Box<Double> myBox3;
```

Виклик узагальненого типу можна розглядати як [виклик звичайного методу](#). При цьому замість передачі аргументу у метод передають аргумент-тип (у даному прикладі: String, Integer та Double).

Зверніть увагу на [пояснення різниці між аргументом-типом та параметром типу](#):

символ T у Foo<T> позначає параметр типу, а String у Foo<String> f — аргумент типу.

У результаті виклику узагальненого типу отримують параметризований тип.

Для створення екземпляру даного класу використовують оператор new, помістивши <тип> між іменем класу та круглими дужками:

```
Box<String> myBox1 = new Box<String>();
```

Diamond

Починаючи з Java 7 і вище можна замінити набір аргументів-типів, необхідних для виклику конструктора узагальненого класу порожнім набором аргументів-типів (<>), оскільки компілятор може визначити їх з контексту. Пару кутових дужок <> називають діамантовим оператором:

```
Box<String> myBox1 = new Box<>();  
Box<Integer> myBox2 = new Box<>();  
Box<Double> myBox3 = new Box<>();
```

Детальніше про [Diamond](#).

Множинні параметри типу

Створимо ArrayList з пар елементів, кожний із яких може мати довільний тип. Пару елементів подамо як окремий generic-клас.

```
import java.util.ArrayList;
import java.util.List;
class Box<T1, T2> {
    private T1 value1;
    private T2 value2;
    Box(T1 value1, T2 value2) {
        this.value1 = value1;
        this.value2 = value2;
    }
    public T1 getValue1() {
        return value1;
    }
    public T2 getValue2() {
        return value2;
    }
}
public class Test {
    public static void main(String[] args) {
        Box<Integer, String> var1 = new Box<>(1, "Apple");
        int vBox1Val1 = var1.getValue1();
        String vBox1Val2 = var1.getValue2();
        System.out.println(vBox1Val1 + " " + vBox1Val2);
        Box<Double, Double> var2 = new Box<>(1.5, 5.5);
        Double vBox2Val1 = var2.getValue1();
        Double vBox2Val2 = var2.getValue2();
        System.out.println(vBox2Val1 + " " + vBox2Val2);
    }
}
```

Результат:

1 Apple
1.5 5.5

Описане вище стосується і інтерфейсів.

Generic-методи

Узагальненими (generic)-методами називають методи із власними параметрами типу. На відміну від узагальнених типів область видимості параметрів типу у даному випадкові визначається generic-методами, у яких вони оголошені.

Синтаксис generic-методу включає параметр типу усередині кутових дужок, котрий знаходиться перед типом, що повертає метод.

Розглянемо [приклад](#):

```
class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
            p1.getValue().equals(p2.getValue());
    }
}
class Pair<K, V> {
    private K key;
    private V value;
    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey() { return key; }
    public V getValue() { return value; }
}
public class Test {
    public static void main(String[] args) {
        Pair<Integer, String> p1 = new Pair<>(1, "apple");
        Pair<Integer, String> p2 = new Pair<>(2, "pear");
        boolean same = Util.<Integer, String>compare(p1, p2);
        System.out.println(same);
        Pair<Integer, String> p3 = new Pair<>(1, "apple");
        Pair<Integer, String> p4 = new Pair<>(2, "pear");
        same = Util.compare(p3, p4);
        System.out.println(same);
    }
}
```

Результат:

false
false

Зверніть увагу на виклик методу `boolean same = Util.<Integer, String>compare(p1, p2);`. Компілятор може визначити типи аргументів, підставлених у відповідні параметри типу, тому немає потреби вказувати їх явно: `same = Util.compare(p3, p4);`.

Обмеження параметрів типу

Часом виникає потреба обмежити типи, котрі можуть бути використані як аргументи типу одним чи кількома значеннями. Наприклад, метод, що оперує числами, може вимагати як параметри тільки екземпляри `Number` чи його дочірніх класів.

Для оголошення обмеженого параметру типу після імені параметру типу подають ключове слово `extends`, після якого вказують верхню межу типу. Наприклад, у даному випадкові — `Number`.

Зауважте, що у даному контексті під `extends` розуміють як "extends" (як у випадкові класів), так і "implements" (як для інтерфейсів). Тобто, у будь-якому разі використовують ключове слово `extends`.

Розглянемо [приклад](#):

```
public class Box<T> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // помилка: параметр – String!
    }
}
```

Такий код не скомпілюється, оскільки виклик `integerBox.inspect("some text");` містить невірний параметр: подано `String` замість [Number](#).

Змінивши "some text" на, до прикладу, 1.5, отримаєм:

T: java.lang.Integer

U: java.lang.Double