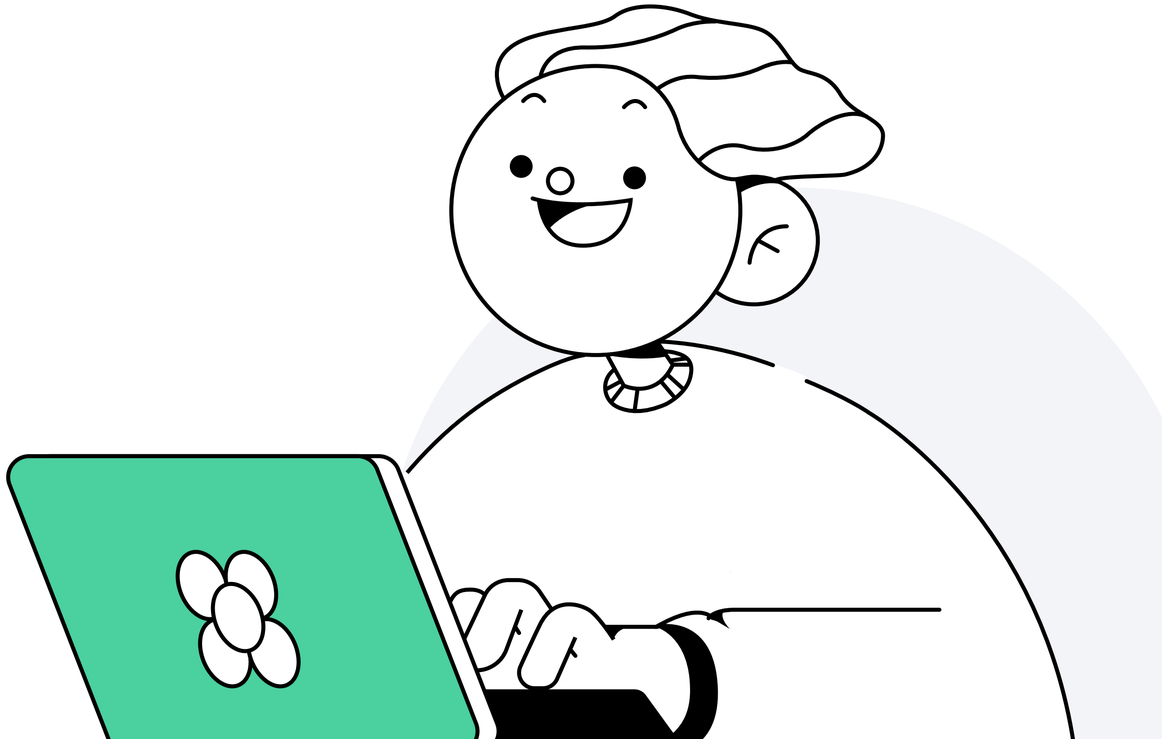
Generics в коллекциях и методах

План занятия

1. Generic-классы и методы
2. Generic и наследования



Несложная задача

Создадим класс Memory, объекты которого способны принимать другие объекты типа String и отвечают на вопрос, какой последний объект был в них сохранён

```
public class Memory {
    protected String value;

    public void save(String value) {
        this.value = value;
    }

    public String getLast() {
        return value;
    }
}
```

Несложная задача

Воспользуемся этим классом и создадим объект **Memory:**

```
public class Main {

    public static void main(String[] args) {

        Memory memory = new Memory();
        memory.save("Petya");
        memory.save("Olya");
        memory.save("Tanya");
        System.out.println(memory.getLast());
    }
}
```

Все ещё несложная задача

Представим, что мы хотим использовать класс Memory не только для String, но и для других объектов.

Решение создать по копии класса Memory для каждого объекта не подходит, так как приведёт к дублированию кода.

Второй вариант — ослабить типы, используемые нами в классе Memory, и вместо String выводить Object, который подойдёт под любое значение.

Однако это решение приведёт к проблеме

```
public class Memory {
    protected Object value;

    public void save(Object value) {
        this.value = value;
    }

    public Object getLast() {
        return value;
    }
}
```

Задача сложнее, чем кажется

Проблема текущего решения в том, что при сохранении последнего значения в переменную метод **getValue(),** который возвращает тип **Object,** потенциально может вернуть нам значение любого типа и, чтобы использовать значение, нужно каждый раз явно приводить тип, а это неправильно:

```
public class Main {

    public static void main(String[] args) {

        Memory memory = new Memory();
        memory.save("Petya");
        memory.save("Olya");
        memory.save("Tanya");
        String value = (String) memory.getLast ();
    }
}
```

Задача сложнее, чем кажется

С этой задачей нам помогут справиться **обобщения — generics** (англ.) или, как часто говорят, **дженерики.** С их помощью достаточно указать, какой тип данных будет храниться в классе **Memory** при создании его экземпляра. И такой класс **Memory** потребуется всего **один**

Код без дженериков

Код с дженериками

Generic-классы и методы

Определение

Обобщения (generics) — средство языка Java для реализации обобщённого программирования.

- Позволяют работать с различными типами данных в одних и тех же методах и классах без изменения их описания.
- Используются для более сильной **проверки типов во время компиляции** и для устранения необходимости в явном приведении типов

Дженерики легко распознать по наличию указателя типа **<T>,** будь то класс, метод или конструктор класса. Через запятую можно указать ещё один тип **<T, U>**

```
public class Memory<T> {

}
```

Сложная задача с простым решением

Давайте перепишем класс **Memory**, используя полученные знания о дженериках.

Можно удобно создавать новые объекты из класса **Memory** и класть объекты любого типа в качестве значения поля **value,** но этот тип должен соответствовать типу **T** класса **Memory.**

Именно в таких ситуациях удобно использовать типизированные классы.

Для метода **getValue** мы указали **T** в качестве типа возвращаемого значения, так как в момент описания этот тип неизвестен (обобщён)

```
public class Memory<T> {
    protected T value;

    public void save(T value) {
        this.value = value;
    }

    public T getLast() {
        return value;
    }
}
```

Создание экземпляра типизированного класса

Теперь можно использовать класс **Memory** гораздо эффективнее:

При создании экземпляра класса **Memory** после имени класса в угловых скобках нужно указать, какой именно тип будет использоваться вместо типизированного параметра, — String.

При таком написании метода Java сам проследит за тем, что тип возвращаемого значения всегда будет соответствовать указанному типу

```
public class Main {

    public static void main(String[] args) {

        Memory<String> memory = new Memory<String>();
        memory.save("Petya");
        memory.save("Olya");
        memory.save("Tanya");
        String value = memory.getLast ();
    }
}
```

Обратите внимание

Для указания типа дженериков используются только классы. Примитивные типы использовать нельзя. Таким образом, **нельзя использовать типы int, double и т. п.**

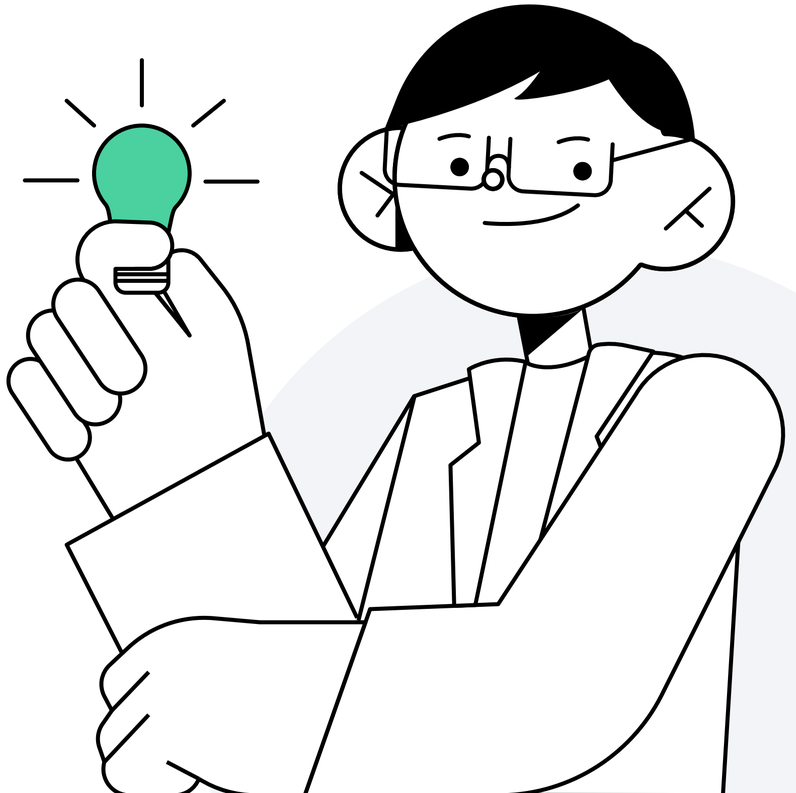
Вместо примитивных типов надо использовать классы-обёртки:

- **Integer** вместо int,
- **Double** вместо double и т. д.

Например, **Memory<Integer>** или **Memory<Double>**

Что можно обобщать с помощью <generics>

- Классы
- Методы
- Конструкторы
- Интерфейсы



Типизированный метод

Создадим метод **choose,** принимающий три параметра. Проблемы начнутся, если мы захотим обобщить этот метод. Применим механизм дженериков:

```
public static void main(String[] args) {
        String v1 = "Petya";
        String v2 = "Olya";
        String result = choose(true, v1, v2);
    }

    public static <T> T choose(boolean flag, T first, T second) {
        if (flag) {
            return first;
        } else {
            return second;
        }
    }
}
```

Типизированный метод

Основное отличие использования дженериков на уровне методов от использования их на уровне классов заключается в том, что чаще всего Java сам вычисляет тип автоматически.

Если мы всё же хотим самостоятельно указать тип, то сделать это нужно следующим образом:

```
String result = Main.<String> choose(true, v1, v2);
```

В случае работы с дженерик-методами мы можем не указывать тип, в случае же с классами это обязательно

Generic
и наследования

Особенности использования Generic вместе с наследованием

Первая особенность: Java запрещает использовать полиморфизм по линии дженериков.

Java не позволяет скомпилировать **Memory<Number> memory = new Memory<Integer>,** иначе это будет означать, что мы без проблем сможем реализовать метод **memory.save(3.5),** передающий дробное число, а это неправильно, так как Integer принимает только целые числа

```
public class Main {

    public static void main(String[] args) {
        Integer i = 100;
        Number n = i;

        Memory<Number> memory = new Memory<Integer>();
        memory.save(3.5);
```

Особенности использования Generic вместе с наследованием

Поэтому существует простое правило:
тип объявления переменной должен соответствовать типу, который вы передаёте объекту.

Справа можно оставить угловые скобки, потому что Java и так понятно, какой тип мы будем использовать

```
public class Main {

    public static void main(String[] args) {
        Integer i = 100;
        Number n = i;

        Memory<Number> memory = new Memory<>();
        memory.save(3.5);
```

Особенности использования Generic вместе с наследованием

Вторая особенность: при указании параметров мы можем конкретизировать, какие типы мы хотим использовать, и таким образом ограничить тип данных, которые можно использовать в нашем методе:

```
public static <T extends Number> T choose(boolean flag, T first, T second) {}
```

Это даёт нам дополнительные возможности: когда мы сообщаем Java дополнительную информацию о параметре, у нас появляются все методы, которые мы можем с ним использовать

Как Generic участвует в наследовании

Создадим типизированный интерфейс Savable для класса Memory, чтобы любой класс, который интерпретирует этот интерфейс, имел объекты типа save:

```
public interface Savable<T> {
   void save(T obj);
}
```

Типизированный интерфейс

При реализации типизированного интерфейса есть две стратегии

1. Типизированный параметр интерфейса задаётся конкретным типом. Тогда класс, реализующий интерфейс, жёстко привязан к типу интерфейса:

```
public class Memory<T> implements Savable<String> {}
```

2. Класс, реализующий интерфейс, типизиуется тем же параметром, что и интерфейс:

```
public class Memory<T> implements Savable<T> {}
```

Использование нескольких обобщённых типов одновременно

Рассмотрим множественную типизацию на примере обобщения класса.

Представляется возможным задать сразу несколько типизированных параметров

```
public class Box<E,T> {
    private E key;
    private T value;

    public Box(E key, T value) {
        this.key = key;
        this.value = value;
    }

    public E getKey() {
        return key;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

Использование нескольких обобщённых типов одновременно

В данном случае для создания экземпляра класса необходимо указать сразу два параметра. Например, **String** и **String:**

```
Box<String, String> box3 = new Box<>("Третья", "Тапочки");
```

Обратите внимание

- Таких параметров может быть множество, но обычно не более двух
- Выбор буквы, описывающей тип, ни на что не влияет.

Отличие сырых и типизированных классов

Свойство	Сырая (raw) коллекция	Типизированная (generics) коллекция
Синтаксис	Box box = new Box();	Box<T> box = new Box<T>();
Безопасность типа	Может содержать любой тип данных. Следовательно, не является типобезопасной	Может содержать только определённый тип данных. Является типобезопасной
Приведение типа	Индивидуальное приведение типа должно быть сделано при каждом поиске	Приведение типов не требуется
Проверка на этапе компиляции	Проверяется на безопасность типов во время выполнения	Проверяется на безопасность типов во время компиляции

Спасибо
за внимание!