Тема: Дженерики в языке программирования Java

1 Слайд: (Начальный экран)
Тема нашей презентации «Дженерики в языке программирования Java»

2 Слайд (Что такое дженерики?)
Дженерики (Generics) — это механизм, позволяющий создавать классы, интерфейсы и методы, которые могут работать с различными типами данных, не теряя безопасности типов во время компиляции.

Как они помогают программистам?
- Позволяют писать более обобщённый код
- Повышают безопасность типов: ошибки с типами данных обнаруживаются на этапе компиляции.
- Снижают необходимость приведения типов (кастинга)

Кастинг — это явное преобразование одного типа данных в другой.
В старых версиях Java программисты часто использовали кастинг при работе с коллекциями. Проблема кастинга заключается в том, что ошибки могут возникать только во время выполнения программы, если тип данных не совпадает. Дженерики устраняют эту проблему, поскольку типы данных проверяются на этапе компиляции.

3 Слайд (Пример использования параметризации и ее работа)
Взгляните на следующий код, он состоит из класса Box и функции main. Main функция демонстрирует пример использования класса Box. Класс Box имеет поле класса value и две функции сеттер и геттер.

В отличи от простых классов, у нас присутствует некоторая переменная T — параметризованный тип, который заменяется на конкретный тип во время компиляции.
Для того, чтобы использовать класс, нам нужно обозначить тип данных, которым он будет оперировать. Затем, мы вызываем сеттер и геттером выводим наше значение. В качестве примера мы вывели значения типов Integer и String. Если бы мы не использовали параметризированный тип, то нам необходимо было создавать два одинаковых по функционалу класса с полями для Integer и String и код бы разросся до двух одинаковых классов
Параметризация позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр
Обеспечивается строгая типизация: ошибки с типами данных ловятся ещё на этапе компиляции.
Выделим некоторые преимущества такого использования:
- Универсальность: Один класс может работать с разными типами данных.
- Повышенная безопасность типов и меньше ошибок.

4 Слайд (Параметризированные интерфейсы)
Параметризация в интерфейсах Java позволяет создавать более гибкие и переиспользуемые структуры. Интерфейсы могут иметь обобщенные параметры, которые позволяют определять методы, переменные и другие характеристики интерфейса для различных типов данных. Это достигается путем объявления типа параметра в угловых скобках. Пример на экране.

Такой интерфейс может быть реализован классами с различными типами данных, что уменьшает дублирование кода и повышает безопасность типов. Использование параметризованных интерфейсов позволяет гарантировать, что передаваемые типы соответствуют ожидаемым, что снижает вероятность ошибок во время выполнения

Преимущества использования параметризации в интерфейсах

- **Гибкость**: Один интерфейс может работать с различными типами данных.
- **Безопасность типов**: Ошибки с типами данных обнаруживаются на этапе компиляции.
- **Снижение дублирования кода**: Позволяет избежать создания нескольких версий одного и того же интерфейса для разных типов данных.

5 Слайд (Параметризированные методы)

Параметризованный метод определяет базовый набор операций, которые будут применяться к разным типам данных, получаемым методом в качестве параметра.

Такие методы позволяют выполнять одну и ту же логику для разных типов данных, что делает код гибким и многократно используемым.

В примере для массива чисел и строк используется таким образом один и тот же метод вывода, что делает код гибким к изменениям логики кода.

6 Слайд (Наследование) /классы/

При наследовании параметризованного класса подкласс может использовать тот же параметр типа или определить свои собственные параметры. Это позволяет создавать специализированные версии базового класса

1. **Параметризованный класс Box<T>**:
   Этот класс принимает параметр типа T, который используется для хранения значения в поле value. Метод getValue() возвращает это значение. Это позволяет создавать экземпляры класса Box для различных типов данных.
2. **Подкласс ExtenedBox<T>:**
   Этот класс наследует от Box<T>, что позволяет ему использовать тот же параметр типа T. Конструктор ExtenedBox принимает значение типа T и передает его в конструктор базового класса с помощью super(value).

7 Слайд (Наследование) /интерфейсы/

Параметризованные интерфейсы могут также наследоваться, позволяя подклассам определять свои параметры.

Принцип действия примера

1. **Параметризованный интерфейс Pair<K, V>:**
   Определяет два метода: getKey() и getValue(), которые возвращают ключ и значение соответственно. Параметры типа K и V позволяют использовать любые типы данных.
2. **Класс OrderedPair<K, V>:**
   Реализует интерфейс Pair и хранит ключ и значение в полях key и value. Конструктор принимает значения для этих полей и сохраняет их.
   Методы getKey() и getValue() возвращают соответствующие значения.
3. **Подкласс CustomPair<K, V>:**
   Наследует от класса OrderedPair и использует те же параметры типа K и V. Конструктор вызывает конструктор базового класса с помощью super(key, value), что позволяет передавать значения ключа и значения.

Эти примеры показывают, как наследование работает в параметризованных классах и интерфейсах. Это позволяет создавать более специфичные классы и интерфейсы, которые используют общую логику, обеспечивая при этом безопасность типов на этапе компиляции.

8 Слайд (Ограничения для generic-типов)

При использовании дженериков в Java важно помнить о ряде ограничений, которые связаны с особенностями реализации и принципами безопасности типов. Вот основные из них:

1. **Невозможность создания экземпляра параметризованного типа через new:**
   Дженерики в Java реализуются через механизм стирания типов (type erasure), что делает невозможным создание объекта T в коде new T(). Это связано с тем, что на этапе выполнения информация о параметре типа отсутствует. 1 пример на экране
2. **Статические поля не могут быть generic-параметрами:**
   Поскольку статические члены класса разделяются всеми экземплярами класса, использование generic-параметров для статических полей невозможно. Каждый экземпляр класса должен иметь собственный параметр типа, что несовместимо со статическими полями. 2 пример на экране
3. **Статические методы не поддерживают generic-параметры:**
   Дженерик-параметры не могут использоваться в сигнатуре статического метода класса, поскольку статические методы не зависят от конкретного экземпляра, следовательно, и от его параметров типа. 3 пример на экране
4. **Generic-типы не могут быть примитивными:**
   Дженерики могут работать только с объектами, а не с примитивными типами (int, double, char и т. д.). Вместо этого следует использовать их обёртки (Integer, Double, Character). 4 пример на экране

9 Слайд (Использование instanceof и ограничений <? extends T> и <? super T>)

1 часть (instanceof)

В параметризованных классах оператор instanceof позволяет проверять тип объекта, но только для базового типа, поскольку информация о параметре типа стирается во время компиляции.

instanceof может использоваться с Wrapper<?> для проверки самого класса (Wrapper), но не его параметризованного типа. Конструкция <?> выступает как «дикий символ» и говорит о неизвестном типе параметра, но сама проверка на конкретный параметр типа (Wrapper<Integer>) невозможна из-за стирания типов.

2 часть <? extends T> и <? super T>)

<? extends T>:

- Ограничивает параметризованный тип, чтобы он был T или его подтипом.
- Используется для чтения данных, но не для их записи, чтобы гарантировать безопасность типов.
- Этот метод (из примера) может принимать List<Integer>, List<Double> и другие списки числовых типов, но не List<String>.

<? super T>:
- Ограничивает параметризованный тип, чтобы он был T или его супертипом.
- Используется для записи данных, когда известно, что тип данных будет суперклассом T.
- Этот метод (из примера) может принимать List<Integer>, List<Number>, List<Object>, но не List<Double>.

<? extends T> и <? super T> позволяют контролировать, какие операции можно безопасно выполнять с параметризованным типом в коллекциях, что особенно полезно при работе с наследуемыми структурами и предотвращении ошибок типов.

10 Слайд (Как использовать дженерики?)
- Коллекции: Дженерики широко применяются в Java коллекциях (List<T>, Map<K, V> и т.д.).
- Обобщённые классы и интерфейсы: Подходят для классов, которым нужно работать с различными типами данных.
- Обобщённые методы: Используются для создания многоразовых функций, например, для обработки массивов или коллекций.

На картинке изображен скриншот с официального сайта Oracle. Который демонстрирует использование Java коллекцию Map. Как мы помним Map может принимать любую пару типов, которую мы зададим. Следовательно в интерфейсе Map используется те самые дженерики K и T, которые позволяют использовать Map для любых типов данных

11 Слайд (Пример избыточного использования)
Когда дженерики будут лишними:
- Статичные типы данных: Если тип данных фиксирован и не изменяется в процессе использования, нет необходимости в дженериках.
- Специфичные классы: Если класс или метод заточены под работу с конкретным типом данных, использование дженериков не оправдано.
- Простые структуры данных: В случае с простыми структурами (например, класс для работы с конкретным типом данных), дженерики добавляют ненужную сложность.
- Избыточное использование дженериков может увеличивать размер байт-кода, что теоретически может снизить производительность на этапе выполнения.

12 Слайд (Аналогии в других языках)
Похожая система Generic Types существует в других языках
Преимущества Java дженериков:
- В Java реализована строгая типизация, обеспечивающая безопасность на уровне компиляции, что делает её реализацию более надёжной по сравнению с C++.
- В отличие от C#, дженерики в Java основаны на стирании типов (type erasure), что снижает нагрузку во время выполнения программы.

На слайде представлен пример использования Generic Types для языков C# и C++

13 Слайд (Заключение)
- Дженерики в Java делают код более гибким, многократно используемым и безопасным.
- Они помогают избегать ошибок на этапе компиляции и делают код более понятным и поддерживаемым.
- В сравнении с другими языками, такими как C# и C++, Java обеспечивает баланс между безопасностью и производительностью.
- Однако, избыточное использование дженериков может привести к усложнению кода, поэтому важно находить баланс.

Список литературы

1. Java Documentation (Oracle) Официальная документация по дженерикам в Java.
https://docs.oracle.com/javase/tutorial/java/generics/index.html
2. "Effective Java" by Joshua Bloch Книга, содержащая лучшие практики программирования на Java, включая использование дженериков.
https://avmim.com/wp-content/uploads/2019/01/Blokh_Dzh_-_Java_Effektivnoe_programmirovanie_2_izdanie_-_2008.pdf
3. "Java: The Complete Reference" by Herbert Schildt Полное руководство по Java, в котором есть разделы, посвящённые дженерикам.
https://djvu.online/file/LKMcAz8DMHvkG
4. GeeksforGeeks Статья о дженериках в Java с примерами.
https://www.geeksforgeeks.org/generics-in-java/
5. Baeldung Статья, подробно объясняющая дженерики и их использование в Java.
https://www.baeldung.com/java-generics
6. "Java Generics and Collections" by Maurice Naftalin and Philip Wadler Книга, посвящённая дженерикам и коллекциям в Java.
https://www.r-5.org/files/books/computers/languages/java/main/Maurice_Naftalin_and_Philip_Wadler-Java_Generics_and_Collections-EN.pdf