**Текст выступления к презентации "Наследование и полиморфизм в Java"**

**(Слайд 1: Титульный лист)**

Докладчик 1: Добрый день, уважаемые слушатели! Сегодня мы, Сагун Александр и Житков Никита, расскажем вам о двух фундаментальных понятиях в Java — наследовании и полиморфизме.

**(Слайд 2: Введение)**

Докладчик 2: Действительно, эти принципы — не просто сухая теория. Это краеугольные камни объектно-ориентированного программирования, которые мы постоянно используем на практике. Именно они позволяют нам грамотно переиспользовать уже написанный код, делая наши программы более структурированными, гибкими и легкими в поддержке.

**(Слайд 3: Основы ООП)**

Докладчик 1: Прежде чем углубиться в детали, давайте вспомним четыре основных принципа ООП. Это:

1. Инкапсуляция — сокрытие внутреннего состояния объекта и деталей реализации.

2. Наследование — механизм, о котором мы подробно поговорим далее.

3. Полиморфизм — второй герой нашего сегодняшнего выступления.

4. Абстракция — использование только тех характеристик объекта, которые важны в текущем контексте.

Сегодня мы сфокусируемся на двух средних пунктах, которые тесно связаны между собой.

**(Слайд 4: Наследование: определение)**

Докладчик 2: Итак, что же такое наследование? Если просто, это механизм, при котором один класс (его называют дочерним, подклассом или производным) может заимствовать поля и методы другого класса (родительского, суперкласса или базового).

Докладчик 1: Отличная аналогия — отношения родителя и ребенка. Ребенок наследует какие-то черты родителя, но при этом он — уникальная личность со своими особенностями. В программировании например: класс `Собака` наследует от класса `Животное`. Всё, что умеет делать животное (например, есть, спать, издавать звук), умеет и собака, но звук она издает специфический — лает.

**(Слайд 5: Пример наследования)**

Докладчик 2: Давайте посмотрим на код. У нас есть базовый класс `Animal` с методом `makeSound()`. От него мы создаем два дочерних класса: `Dog` и `Cat`. Обратите внимание на ключевое слово `extends` — оно и означает "расширяет". То есть класс `Dog` \*расширяет\* функциональность класса `Animal`.

Докладчик 1: И здесь мы видим важный момент: и собака, и кошка переопределяют метод `makeSound()`, предоставляя свою, конкретную реализацию. Это наш первый шаг к полиморфизму, но о нем чуть позже.

**(Слайд 6: Схема наследования)**

Докладчик 2: Наглядно эту иерархию можно представить в виде такой схемы. Класс `Animal` находится наверху, а классы `Cat` и `Dog` являются его потомками. Это классическое дерево наследования по принципу "is-a" (является): "Собака \*является\* животным", "Кошка \*является\* животным".

**(Слайд 7: Преимущества наследования)**

Докладчик 1: Какая же от этого практическая польза? Во-первых, это повторное использование кода. Нам не нужно в классах `Dog` и `Cat` заново описывать общие для всех животных действия, например, метод `eat()`. Мы пишем его один раз в базовом классе.

Докладчик 2: Во-вторых, это удобство расширения и сопровождения. Если нам нужно добавить новое животное, например, `Bird`, мы просто создаем новый класс, расширяющий `Animal`, и реализуем его уникальное поведение. А если нам нужно изменить общую логику для всех животных, мы вносим правки всего в одном месте — в базовом классе.

**(Слайд 8: Ограничения наследования)**

Докладчик 1: Однако у наследования в Java есть важное ограничение — множественное наследование не поддерживается. Класс может наследовать только от одного родительского класса. Это было сделано для избежания ambiguity, неоднозначности.

Докладчик 2: Представьте, что класс `RobotDog` наследует и от `Dog`, и от `Robot`, и в обоих родителях есть метод `charge()` ‘run’ . Откуда его брать? Чтобы решить эту проблему, в Java используют интерфейсы, которые позволяют реализовывать множественное "наследование" поведения.

**(Слайд 9: Полиморфизм: определение)**

Докладчик 2: Переходим ко второй части — полиморфизму. Если дословно, "поли" — много, "морф" — форма. То есть возможность одного интерфейса иметь много реализаций.

Докладчик 1: Простая жизненная аналогия — пульт дистанционного управления. Один и тот же пульт с кнопкой "включить" может управлять телевизором, кондиционером или музыкальным центром. Кнопка одна, а результат ее нажатия — разный, в зависимости от того, на какой прибор он направлен.

**(Слайд 10: Виды полиморфизма)**

Докладчик 2: В Java полиморфизм бывает двух видов.

Первый — статический, или полиморфизм времени компиляции. Он достигается через перегрузку методов. Это когда в одном классе есть несколько методов с одним именем, но разными параметрами. Компилятор уже на этапе компиляции определяет, какой именно метод нужно вызвать.

Докладчик 1: Второй вид — динамический, или полиморфизм времени выполнения. Это и есть "тот самый" полиморфизм в ООП. Он достигается через переопределение методов в дочерних классах, и решение о том, какой метод вызывать, принимается уже во время работы программы.

**(Слайд 11: Пример перегрузки методов)**

Докладчик 1: Вот наглядный пример перегрузки. У нас класс `Calculator` с двумя методами `add`. Один складывает целые числа, другой — числа с плавающей точкой. Мы просто вызываем `calc.add(...)`, а Java сама выбирает нужную версию метода в зависимости от переданных аргументов.

**(Слайд 12: Пример переопределения методов)**

Докладчик 2: А вот здесь в игру вступает динамический полиморфизм. Мы создаем переменную `a` типа `Animal`, но помещаем в нее объект типа `Dog`. И когда мы вызываем `a.makeSound()`, выполняется не метод родительского класса `Animal`, а метод объекта `Dog`.

Докладчик 1: Именно это и есть мощь полиморфизма! JVM смотрит не на тип ссылки (`Animal`), а на тип самого объекта в памяти (`Dog`) и вызывает метод именно этого, реального объекта. Это называется поздним связыванием.

**(Слайд 13: Схема полиморфизма)**

Докладчик 2: Эта схема суммирует идею. У нас есть общий тип — `Animal`. И у нас есть множество конкретных реализаций. Когда мы через общую ссылку на `Animal` вызываем `makeSound()`, система сама находит правильную реализацию в зависимости от того, какой именно объект находится "под капотом".

**(Слайд 14: Зачем нужен полиморфизм?)**

Докладчик 1: Зачем все это нужно? Полиморфизм позволяет писать универсальный и гибкий код. Представьте метод `vet(Animal a)`, который принимает любое животное. Нам не нужно писать отдельные методы `vet(Dog d)`, `vet(Cat c)`. Мы пишем один, и он будет работать с любым будущим потомком `Animal`, даже с тем, которого мы еще не придумали.

Докладчик 2: Это делает архитектуру программы масштабируемой. Мы можем легко добавлять новые сущности, не переписывая существующий код, который с ними работает. Это краеугольный камень многих современных фреймворков.

**(Слайд 15: Комбинация наследования и полиморфизма)**

Докладчик 1: Теперь давайте посмотрим, как эти две концепции работают вместе в более зрелом примере. У нас есть абстрактный класс `Shape`. Ключевое слово `abstract` говорит, что мы не можем создать объект "фигура", это слишком общее понятие. Но мы можем обязать всех потомков реализовать метод `draw()`.

Докладчик 2: Классы `Circle` и `Rectangle` наследуют от `Shape` и переопределяют метод `draw()`. И вот полиморфизм в действии: мы можем создать список `List<Shape> shapes`, добавить туда круги и прямоугольники, и в цикле вызвать `shape.draw()` для каждого. Каждая фигура нарисуется правильно, своим способом.

**(Слайд 16: Пример из жизни)**

Докладчик 2: И, наконец, реальный пример из жизни — платежные системы. У нас есть интерфейс `Payment` с методом `pay()`. Разные классы реализуют этот интерфейс по-своему: оплата картой, через PayPal, криптовалютой.

Докладчик 1: В коде нашего приложения мы работаем только с интерфейсом `Payment`. Мы можем написать `Payment p = new CardPayment(); p.pay(100);`. Завтра, если мы захотим добавить оплату через Apple Pay, нам нужно будет лишь создать новый класс, реализующий `Payment`, и подставить его. Основной код бизнес-логики останется неизменным. Это и есть сила полиморфизма.

**(Слайд 17: Источники)**

Докладчик 1: Для более глубокого погружения в тему мы использовали и рекомендуем следующие источники: официальная документация Oracle, GeeksforGeeks, W3Schools и другие.

**(Завершение)**

Докладчик 2: Спасибо за внимание! Мы рады ответить на ваши вопросы.

# Дополнение: Спецификаторы доступа и наследование (Саша)

В Java существуют четыре основных спецификатора доступа: public, protected, default (package-private) и private. Они определяют, где можно использовать члены класса (поля, методы, конструкторы) и влияют на то, что именно будет доступно при наследовании.

• public — доступен всем классам в любом пакете.  
• protected — доступен только классам из того же пакета и всем подклассам.  
• default (без модификатора) — доступен только внутри того же пакета.  
• private — доступен только внутри самого класса и не наследуется.

Пример, показывающий, какие методы доступны наследнику:

class Parent {  
 private void method1() {}  
 protected void method2() {}  
 public void method3() {}  
}  
  
class Child extends Parent {  
 void test() {  
 // method1(); // Ошибка — private не наследуется  
 method2(); // Разрешено — protected доступен  
 method3(); // Разрешено — public доступен всем  
 }  
}

# Дополнение: Аннотации в контексте наследования и полиморфизма (Никита)

Аннотации — это метатеги, а по сути дополнительные модификаторы, которые добавляются к коду и применяются к объявлению пакетов, классов, конструкторов, методов, полей, параметров и локальных переменных.

Аннотации помогают компилятору и разработчикам понимать, как используется тот или иной элемент кода. В контексте наследования и полиморфизма особенно важны аннотации @Override, @Inherited и @Deprecated.

• @Override — указывает, что метод переопределяет метод родительского класса. Помогает избежать ошибок при несоответствии сигнатур методов.  
• @Inherited — позволяет аннотациям наследоваться дочерними классами. Если суперкласс помечен аннотацией с @Inherited, подклассы будут также считаться аннотированными.  
• @Deprecated — обозначает, что элемент устарел и его не рекомендуется использовать.

Пример использования @Override:

class A {

public static void method() { … }

}

class B extends A {

@Override

public static void method() { … }

}

Пример использования @Inherited:

// Обычная аннотация, без @Inherited

@Retention(RetentionPolicy.RUNTIME)

@interface Info {}

@Info

class Animal { }

class Dog extends Animal { }

public class Main {

public static void main(String[] args) {

System.out.println(Animal.class.isAnnotationPresent(Info.class)); // true

System.out.println(Dog.class.isAnnotationPresent(Info.class)); // false

}

}

Хотя класс Dog наследует Animal, аннотация @Info не передалась ему Теперь добавим @Inherited к аннотации Info:

// Добавляем @Inherited

@Inherited

@Retention(RetentionPolicy.RUNTIME)

@interface Info {}

@Info

class Animal { }

class Dog extends Animal { }

public class Main {

public static void main(String[] args) {

System.out.println(Animal.class.isAnnotationPresent(Info.class)); // true

System.out.println(Dog.class.isAnnotationPresent(Info.class)); // true

}

}

Пример использования @Deprecated:

class Calculator {

@Deprecated

int addOld(int a, int b) { return a + b; }

int add(int a, int b) { // Новый метод return Math.addExact(a, b); }

}

public class Main {

public static void main(String[] args) { Calculator c = new Calculator(); //Предупреждение: метод устарел

c.addOld(2, 3); }

}

# Дополнение: Ковариантные типы возвращаемого значения (Саша + Никита что-то (?))

Ковариантные типы позволяют переопределённому методу возвращать более конкретный тип, чем метод родителя. Это делает код гибче и позволяет использовать наследование более естественно.

Пример: если базовый метод возвращает объект типа Animal, то переопределённый метод может возвращать объект потомка, например Dog. Это допустимо, так как Dog — это подтип Animal.

class Animal {}  
class Dog extends Animal {}  
  
class AnimalFactory {  
 Animal create() { return new Animal(); }  
}  
  
class DogFactory extends AnimalFactory {  
 @Override  
 Dog create() { return new Dog(); }  
}

Таким образом, ковариантные типы обеспечивают совместимость между методами родителя и наследника, улучшая читаемость и безопасность кода.