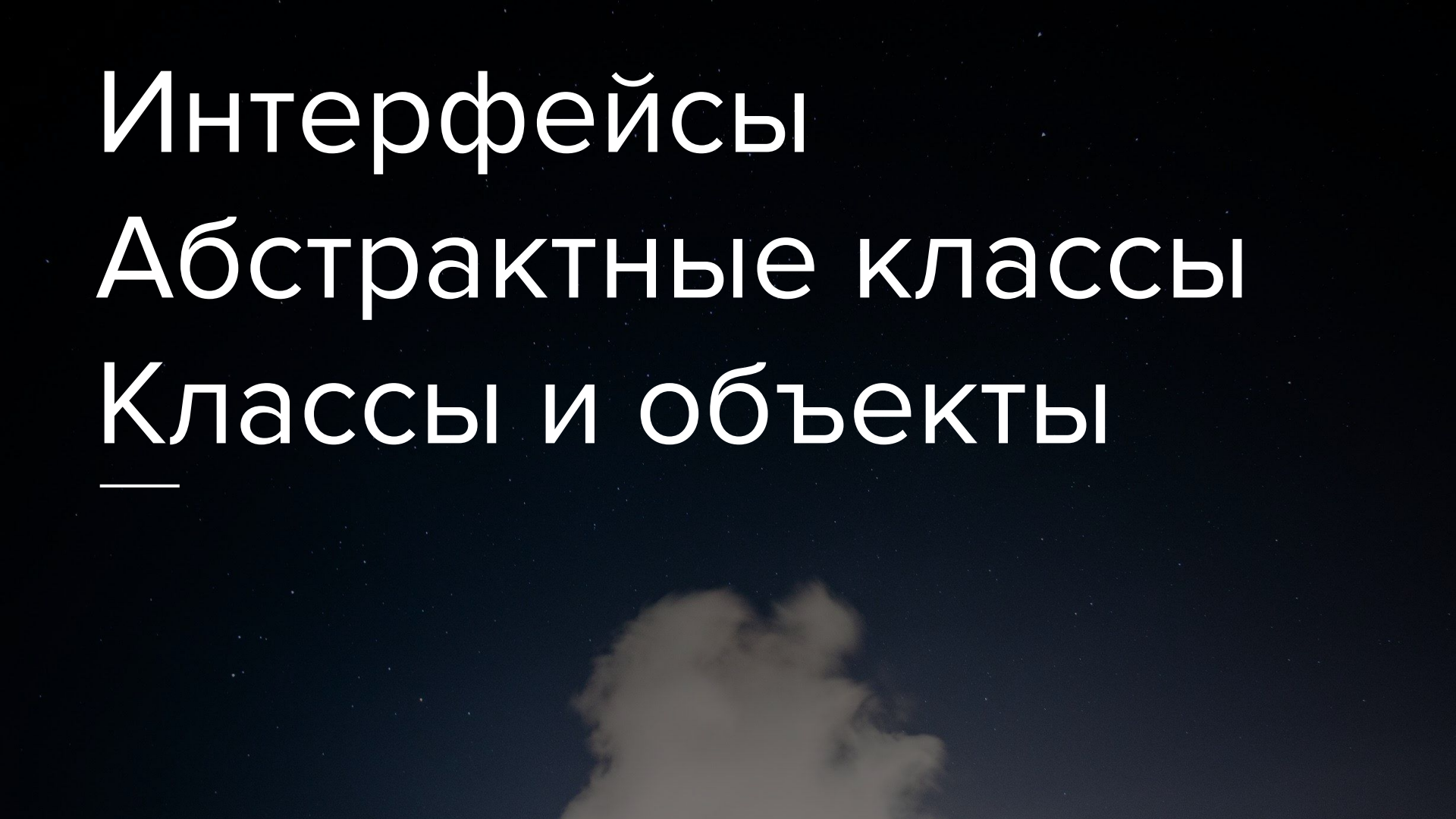
Интерфейсы
Абстрактные классы
Классы и объекты



Java — объектно-ориентированный язык, это означает, что все структуры данных, кроме примитивов, - объекты.

Программы строятся на взаимодействиях объектов, объекты могут взаимодействовать между собой с помощью методов.

В Java **объекты описываются с помощью классов, в которых указываются все возможные методы и свойства**. Поэтому **создание объекта всегда начинается с написания его класса**.

Организация классов в пакеты

В Java обычно каждый класс хранится в отдельном исходном файле (*.java), и для их удобной организации и разрешения конфликтов имен применяются пакеты.

Пакет — это каталог с классами.
Пакеты часто именуются как доменное имя компании наоборот (начиная с корневого домена).

Например, класс BattleUnit лежит в пакете com.ifmo.battle

Организация классов в пакеты

Таким образом, полное имя класса будет содержать название пакета: com.ifmo.battle.BattleUnit. Это позволяет иметь множество классов с именем BattleUnit, но в разных пакетах, например, org.nicegame.unit.BattleUnit.

Но пользоваться такими длинными именами очень неудобно, поэтому используют ключевое слово import, которое позволяет убрать названия пакетов из имени класса.

Организация классов в пакеты

Полное имя:

```
public class ObjectsExample1 {
   public static void main(String[] args) {
      com.itmo.battle.BattleUnit unit= new com.itmo.battle.BattleUnit(100, 20);
   }
}
```

С использованием импорта:

```
import com.itmo.battle.BattleUnit;
public class ObjectsExample1 {
   public static void main(String[] args) {
      BattleUnit unit = new BattleUnit(100, 20);
   }
}
```

Создание классов и объектов

Но прежде, чем создать объект BattleUnit необходимо создать шаблон, по которому он будет создан, описать его структуру.
Для этого необходимо **создать класс**.

```
public class BattleUnit {

}
```

указание на то,
что мы создаем класс

После того, как класс создан, можно **создавать объекты**

```
public static void main(String[] args) {

   BattleUnit  unit = new BattleUnit();

}
```

Создание классов и объектов

Для того, чтобы объект обладал какими либо характеристиками и имел возможность совершать действия, их необходимо прописать в классе.

```
public class BattleUnit {
        int health;
        int attackScore;

        public void setHealth(int health) {
                this.health = health;
        }

        public void setAttackScore(int attackScore) {
                this.attackScore = attackScore;
        }
}
```

свойства (атрибуты)

методы
данные методы позволяют задать значения свойствам health и attackScore

ссылка на текущий объект

Создание классов и объектов

Теперь мы можем создать объект и вызвать перечисленные методы, задав необходимые свойства
(вызов метода или доступ к свойству осуществляется через точку)

```
public static void main(String[] args) {
    BattleUnit unit = new BattleUnit();  // создали объект
    unit.setHealth(100); // задали значение для health
    unit.setAttackScore(20); // задали значение для attackScore
}
```

Создание классов и объектов

У каждого объекта есть **конструктор**, который и вызывается при его создании. **Роль конструктора** — начальная инициализация объекта.

У нашего класса конструктор определен неявно, т. е. при компиляции класс развернется в такую структуру:

```
public class BattleUnit {
   int health;
   int attackScore;

   public BattleUnit(){   }  // конструктор
   // остальные методы
}
```

Создание классов и объектов

Пока конструктор не делает ничего, но перепишем наш класс таким образом, чтобы юнит сразу создавался с указанными здоровьем и очками атаки, те у нас будет возможность установить эти значения на этапе создания (при вызове конструктора).

```
public class BattleUnit {
   int health;
   int attackScore;

   public BattleUnit(int health, int attackScore) {
      this.health = health;
      this.attackScore = attackScore;
   }
 // остальные методы
}
```

При создании объекта, все поля устанавливаются в значения по умолчанию.
Для чисел это 0, для boolean — false, для ссылок на объекты — null.

теперь создание объекта будет выглядеть следующим образом:

```
BattleUnit unit = new BattleUnit(100, 20);
```

создали объект со значением health - 100 и attackScore - 20

Наш класс выглядит следующим образом:

```
public class BattleUnit {
   int health;
   int attackScore;

   public BattleUnit(int health, int attackScore) {
      this.health = health;
      this.attackScore = attackScore;
   }
   public void setHealth(int health) {
      this.health = health;
   }
   public void setAttackScore(int attackScore) {
      this.attackScore = attackScore;
   }
   public int getHealth() {
      return health;
   }
   public int getAttackScore() {
      return attackScore;
   }
   public void attack(BattleUnit enemy){
      enemy.health -= this.attackScore;
   }
}
```

Наследование

Важнейшим принципом ООП является наследование.
Оно позволяет значительно сократить количество дублируемого когда и предоставляет легкую эволюцию объектов.
Суть заключается в том, что дочерний класс наследует свойства и методы родителя.
Таким образом, потомок имеет весь функционал родительского класса без прямого копирования кода, после чего может добавлять свои свойства и функции.

Наследование

Допустим, мы решили создать еще юнитов, но с дополнительными возможностями.

```
public class Knight extends BattleUnit{
   public Knight(int health, int attackScore){
      super(health, attackScore);
   }
   public void addHealth(){
      if (this.health > 1){
         this.health += 5;
      }
   }
   @Override
   public void attack(BattleUnit enemy){
      enemy.health -= this.attackScore;
      addHealth();
   }
}
```

наследование осуществляется при помощи слова **extends**
При этом дочерний класс Knight должен расширять функционал родителя BattleUnit

Для вызова родительского метода в дочернем классе используется слово **super**

Если мы хотим **переопределить метод** родителя мы должны создать точно такой же метод и описать его функционал. Лучше явно указывать на переопределения с помощью @Override

Наследование

Некоторые **правила наследования**:

1. Ключевое слово extends используется в Java для реализации наследования.
2. Private-члены суперкласса недоступные для подклассов. Остальные члены суперкласса доступны для подклассов, методы суперкласса можно расширить или полностью переопределить в дочернем классе
3. Подкласс с уровнем доступа default (по умолчанию) доступен другим подклассам только если они находятся в том же пакете!
4. Конструкторы суперкласса не наследуются подклассами.
5. Если суперкласс не имеет конструктора по умолчанию, то подкласс должен иметь явный конструктор.
6. ava не поддерживает множественное наследование, поэтому подкласс может наследовать только один класс!

Наследование

Теперь мы можем создавать объект на основе нового класса.

```
public static void main(String[] args) {
   Knight knight1 = new Knight(90, 10);
   knight1.getHealth();  // объекту доступны методы родителя
   knight1.getAttackScore();  // объекту доступны методы родителя

   Knight knight2 = new Knight(100, 30);
   knight2.getHealth();   // объекту доступны методы родителя
   knight2.getAttackScore();  // объекту доступны методы родителя

   knight1.attack(knight2); // и свои (либо переопределенные) методы
}
```

Модификаторы доступа

Еще одним **важнейшим принципом ООП является инкапсуляция**. Она позволяет ограничить область видимости методов и полей класса (и даже самих классов). Делается это для того, чтобы скрыть детали реализации классов, т. к. их использование может нарушить логику работы.
Инкапсуляция в Java обеспечивается модификаторами доступа:

- **public** — видим везде;
- **protected** — видим только в рамках пакета, где находится класс, а также наследникам из любого пакета;
- **default** (package-private, без модификатора) — видим только в рамках своего пакета;
- **private** — видим только в рамках своего класса.

Модификаторы доступа

Модификаторы	То же класс	То же пакет	Подкласс	Другие пакеты
public	ДА	ДА	ДА	ДА
protected	ДА	ДА	ДА	НЕТ
default	ДА	ДА	НЕТ	НЕТ
private	ДА	НЕТ	НЕТ	НЕТ

Модификаторы доступа

```
public class Counter {  // Класс доступен везде.
    private int cnt; // Поле cnt инкапсулировано в классе Counter.

  public void increment() {    // Метод доступен везде.
      cnt++;
  }

  public int getValue() {    // Метод доступен везде.
      return cnt;
  }
}

Counter counter = new Counter();
counter.increment();
 int val = counter.getValue();
counter.cnt++; // Ошибка!
```

Private поле не видно и наследникам:

```
public class Counter2 extends Counter {
  @Override
  public void increment() {
      super.cnt += 2; // Ошибка!
  }
}
```

Чтобы поле стало доступно наследникам, модификатор нужно сменить с private на protected.

Абстрактные классы и интерфейсы

Java предоставляет дополнительные инструменты для построения абстрактных моделей, к ним относятся абстрактные классы и интерфейсы.
Абстрактный класс — это класс, который может иметь объявленные, но нереализованные методы. Невозможно создать непосредственно экземпляр такого класса.
Интерфейс, в отличие от абстрактного класса, может иметь только объявления методов и констант, default методы с реализацией (java 8). Так же как и абстрактный класс, нельзя создать экземпляр интерфейса.
Обычные классы, которые наследуют абстрактный класс или реализуют интерфейс обязаны иметь реализацию всех абстрактных/интерфейсных методов. Обычный класс может имплементировать больше одного интерфейса.

Абстрактные классы

Абстрактные классы используются, чтобы создать класс с реализацией метода по умолчанию для подклассов.
Абстрактный класс может иметь как абстрактные методы (не имеют реализации), так и методы с реализацией.

1. abstract — ключевое слово при объявлении класса. Чтобы создать абстрактный класс, нужно дописать ему ключевое слово abstract при объявлении класса.
2. Нельзя создать экземпляр абстрактного класса.
3. Если в классе есть абстрактные методы, то класс также должен быть объявлен абстрактным с помощью ключевого слова abstract

Абстрактные классы

4. Если в абстрактном классе нет ни одного метода с хоть какой-то реализацией, то лучше использовать интерфейс
5. Подкласс абстрактного класса должен реализовать все абстрактные методы, если подкласс сам не является абстрактным классом.
6. Абстрактные классы могут реализовывать интерфейсы, даже не обеспечивая реализацию методов интерфейса.
7. Абстрактные классы являются базой для подклассов, которые реализуют абстрактные методы и переопределяют или используют реализованные методы абстрактного класса.

Интерфейсы

Интерфейсы обеспечивают абстракцию в Java.
Интерфейс обеспечивает контракт для всех классов, которые его реализуют.
Интерфейсы хороши для создания начальной точки и создания иерархии в проекте.

1. **interface** — ключевое слово для создания интерфейса.
2. создать экземпляр интерфейса в Java нельзя.
3. интерфейс обеспечивает абсолютную абстракцию.
4. интерфейсы не могут иметь конструкторов.
5. по умолчанию любой атрибут интерфейса является public, static и final
6. по умолчанию методы интерфейса неявно abstract и public
7. **модификатор private** доступен, начиная с Java 9
8. **static методы с реализацией**, доступны, начиная с Java 8

Интерфейсы

9. **default методы с реализацией**, доступны, начиная с Java 8 (такие методы можно не переопределять)
10. интерфейс может реализовать другой интерфейс. public interface Figures extends SomeInterface{} - пример интерфейса, который наследует другой интерфейс.
11. Java обеспечивает множественное наследование интерфейсов — это означает, что интерфейс может наследовать несколько интерфейсов.
12. ключевое слово **implements** используется классами для реализации интерфейса.
13. класс, реализующий интерфейс, должен обеспечить реализацию всех его методов, если только это не абстрактный класс