Прикладное программирование

Лекция №3.

1. Статическое и динамическое связывание методов (повтор);
2. Передача и изменение аргументов методов класса (повтор);
3. Наследование;
4. Множественное наследование;
5. Интерфейсы;
6. Композиция;
7. Вложенные классы.

Статическое и динамическое связывание методов

Модификаторы – зарезервированные слова.

- **private, protected, public –** правила доступа к методу.
- **static –** принадлежность к общим методам класса.
- **final –** невозможность переопределения методов в потомках.
- **native –** метод задан во внешней библиотеке на другом языке (DLL).
- **abstract –** абстрактный метод, не имеет реализации.
- **synchronized –** синхронизация при работе с потоками.

Статическое и динамическое связывание методов

- **static/final** – при компиляции кода действует **статическое связывание.**

Что это значит?

В скомпилированный код помещается ссылка на метод именно того класса, чье имя указано в исходном коде.

Т.е. в месте вызова метода происходит связывание имени метода с его исполняемым кодом.

Это **раннее связывание**, т.к. происходит на этапе компиляции выполняемой программы.

Во всех остальных случаях в Java выполняется **позднее связывание**, т.к. происходит на этапе выполнения программы непосредственно во время вызова метода.

Передача и изменение аргументов методов класса

В Java применяется передача аргументов метода **по значению.**

Т.о. в Java есть ряд особенностей при работе с входными параметрами:

- метод **не может** изменить значение параметра примитивного типа

(число, Boolean);

- метод **не может** изменить параметр ссылочного типа так, чтобы он ссылался на другой объект;
- метод **может** изменить внутреннее значение параметра ссылочного (т.е. объектного) типа с помощью предложенных тем методов.

ОТЛИЧИЯ:

- **C++ позволяет передавать аргументы по значению и по ссылке.**
- **Т.к. C++ позволяет работать с указателями, то при передаче по значению можно использовать в качестве параметров указатели на области памяти, где располагаются данные, и изменять их с помощью операции разыменования.**
- **Java позволяет передавать аргументы только по значению. Указатели в Java не допускаются.**

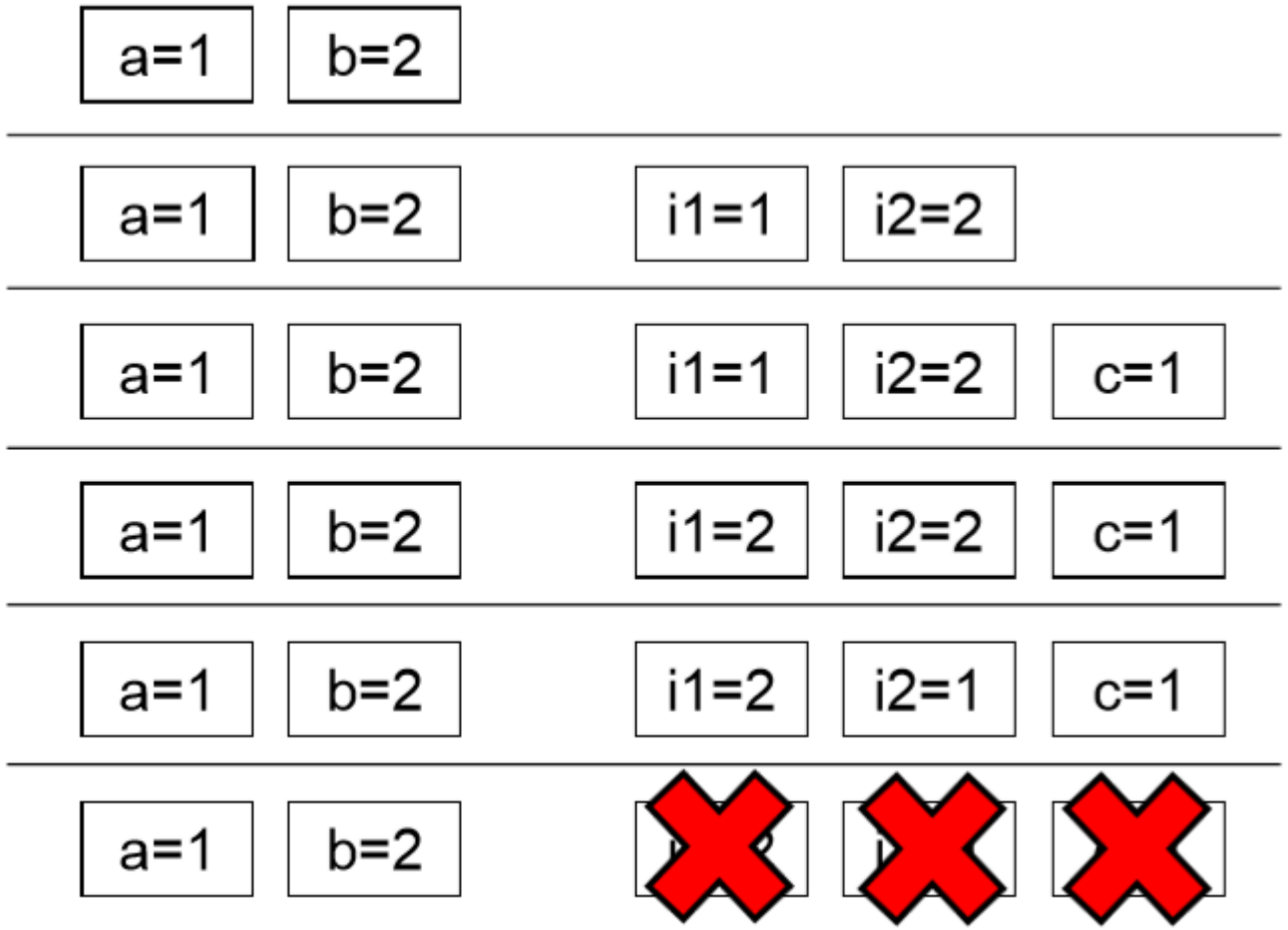
Пример (решение 1): поменять местами две переменных

Задача: реализовать метод, меняющий местами значения двух целочисленных переменных.

Решение 1:

```
class MyClass {
    static void swap(int i1, int i2) {
        int c = i1;
        i1 = i2;
        i2 = c;
    }
    public static void main(String[] args) {
        int a = 1;
        int b = 2;
        swap(a, b);
        System.out.println("a="+a+", b="+b);
    }
}
```

Пример (решение 1): как это должно работать




```
int a=1; int b=2;

swap(a,b) {
    int c = i1;
    i1 = i2;
    i2 = c;
}
```

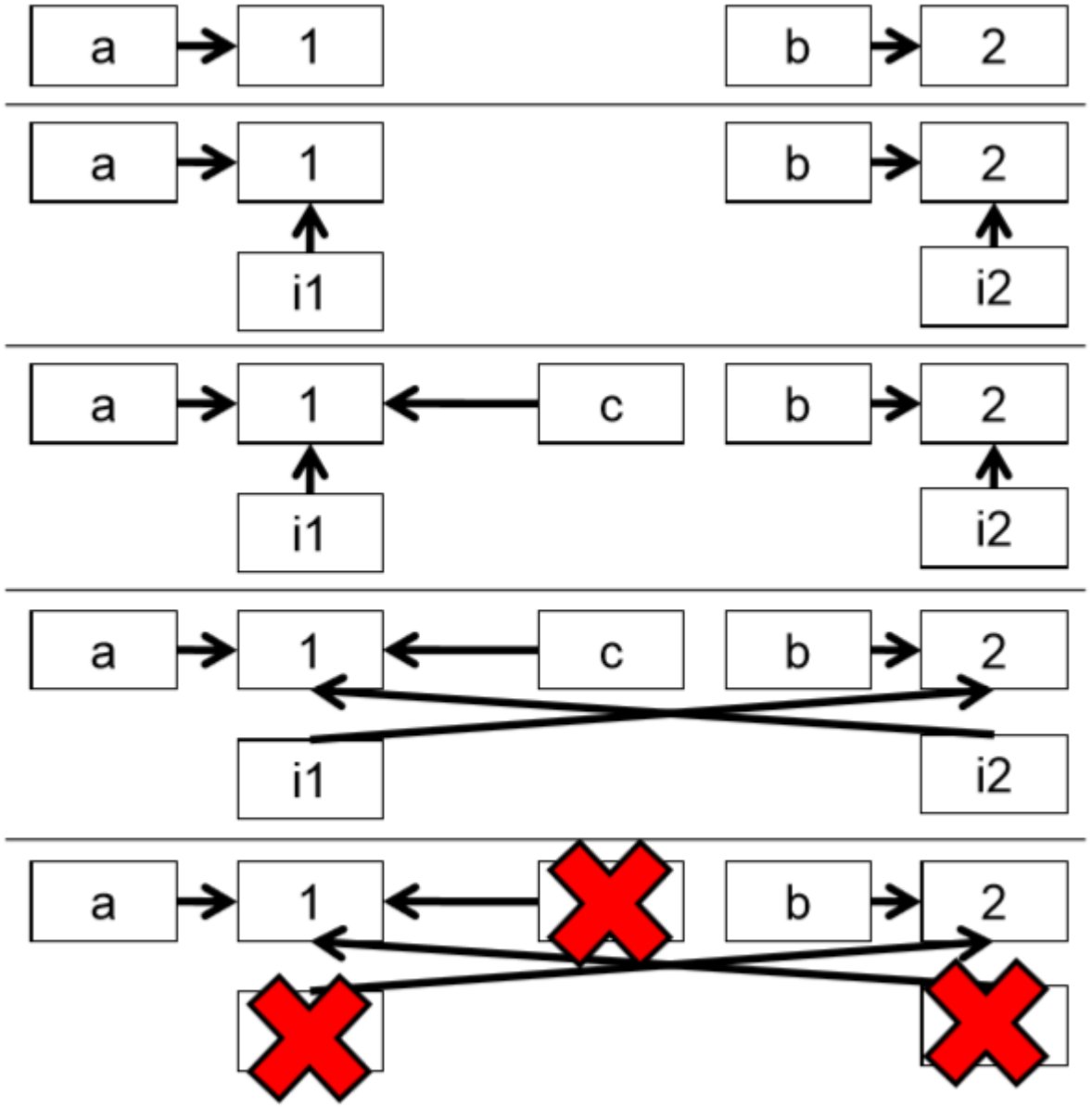
Пример (решение 2): поменять местами две переменных

Возможно все потому, что значения примитивных типов передаются по значению. А давайте теперь использовать типы-оболочки Integer, которые являются ссылочными. Теперь все точно будет тип-топ!

Решение 2:

```
class MyClass {
    static void swap(Integer i1, Integer i2) {
        Integer c = i1;
        i1 = i2;
        i2 = c;
    }
    public static void main(String[] args) {
        Integer a = 1;
        Integer b = 2;
        swap(a, b);
        System.out.println("a="+a+", b="+b);
    }
}
```

Пример (решение 2): как это должно работать




```
Integer a=1;
Integer b=2;

swap(a,b) {

    Integer c = i1;

    i1 = i2;
    i2 = c;

}
```

Пример (решение 3): поменять местами две переменных

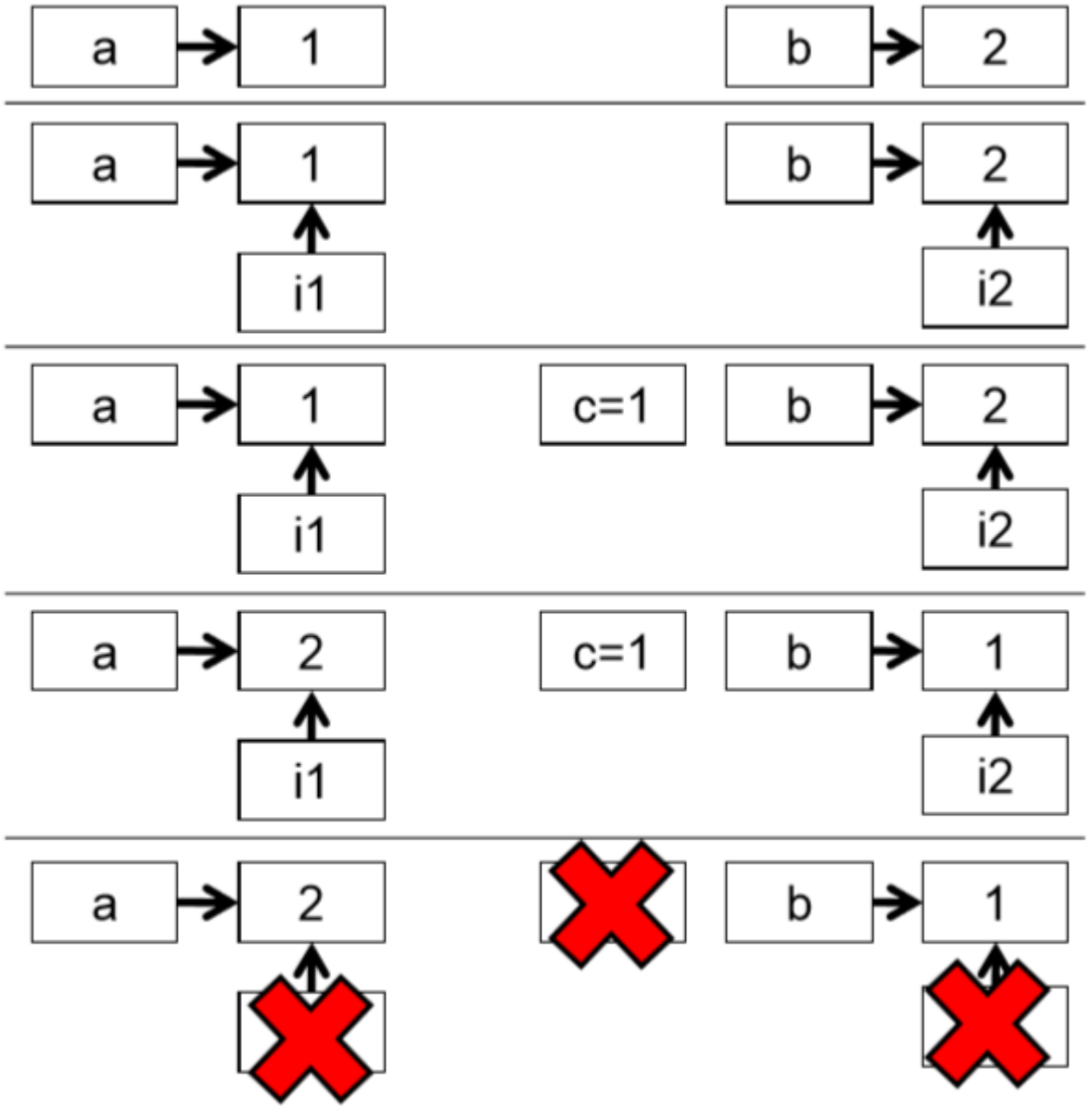
Единственный способ достигнуть задуманное – попросить объект самостоятельно изменить свои данные.

К сожалению, в стандартных классах-оболочках для примитивных данных такой возможности не предоставляется.

Пример (решение 3): поменять местами две переменных

```
class MyInteger {
    private int i;
    public MyInteger(int i) { this.i = i; }
    public int getI() { return i; }
    public void setI(int newI) { i = newI; }
}
class MyClass {
    static void swap(MyInteger i1, MyInteger i2) {
        int c = i1.getI();
        i1.setI(i2.getI());
        i2.setI(c);
    }
    public static void main(String[] args) {
        MyInteger a = new MyInteger(1);
        MyInteger b = new MyInteger(2);
        swap(a, b); // Теперь в a - 2, а в b - 1
    }
}
```

Пример (решение 3): как это должно работать




```
MyInteger a =
    new MyInteger(1);
MyInteger b =
    new MyInteger(2);

 swap(a,b)  {

        int c = i1.getI();

        i1.setI(i2.getI());
        i2.setI(c);

}
```

Наследование в Java

Наследование – отношение между классами, при котором характеристики суперкласса передаются подклассу без необходимости их повторного определения.

Как и в C++, в Java потомок наследует от предка все поля данных и методы, хотя помеченные как **private** напрямую недоступны.

Правила хорошего тона требуют помечать поля данных как **private**, а доступ к ним обеспечивать с помощью методов **get (селектор)** и **set (модификатор)**.

ЗАЧЕМ?

Т.к. прямой доступ к полям данных **угрожает целостности объекта.**

Наследование в Java

Подкласс дополняет члены суперкласса своими полями и методами.

Если имена методов совпадают, а параметры отличаются, это называется **перегрузка методов (статический полиморфизм).**

Если имена и параметры методов совпадают – **динамический полиморфизм.**

Т.о. в подклассе можно переопределить метод с таким же именем, списком параметров и возвращаемым значением, что и у суперкласса.

Способность ссылки динамически определять версию переопределяемого метода в зависимости от переданного в сообщении этой ссылке типа объекта и есть **полиморфизм,**

что в свою очередь является основой для реализации механизма **динамического (позднего) связывания.**

Наследование в Java

Определение полей с теми же самыми именами, как и переопределение методов, приводит к **затемнению полей или методов в классе-потомке *(утеря видимости).***

Чтобы обратиться к затененному свойству используют записи:

- **super**.<Имя_поля>;
- **super**.<Имя_метода>(<Список_параметров>).

Использовать вызовы с помощью ключевого слова **super** разрешается только для конструкторов, а также методов и полей данных объектов.

Комбинации вида **super.super**.<Имя_поля_или_метода> запрещены.

Обращения с помощью ключевого слова **super** для методов и переменных класса, объявленных как ***static***, также запрещены.

this.* - если в методе объявлены локальные переменные с одинаковым именем, что и переменные экземпляра класса.

Множественное наследование

Множественное наследование: класс может иметь бесчисленное количество предков.

Зачем оно нужно?

1. приходится совмещать в объекте поведение, которое характерно для нескольких независимых иерархий;
2. необходимо писать единый полиморфный код для объектов из подобных иерархий.

Следовательно: необходимо вызывать методы с одинаковой сигнатурой, но разной реализацией.

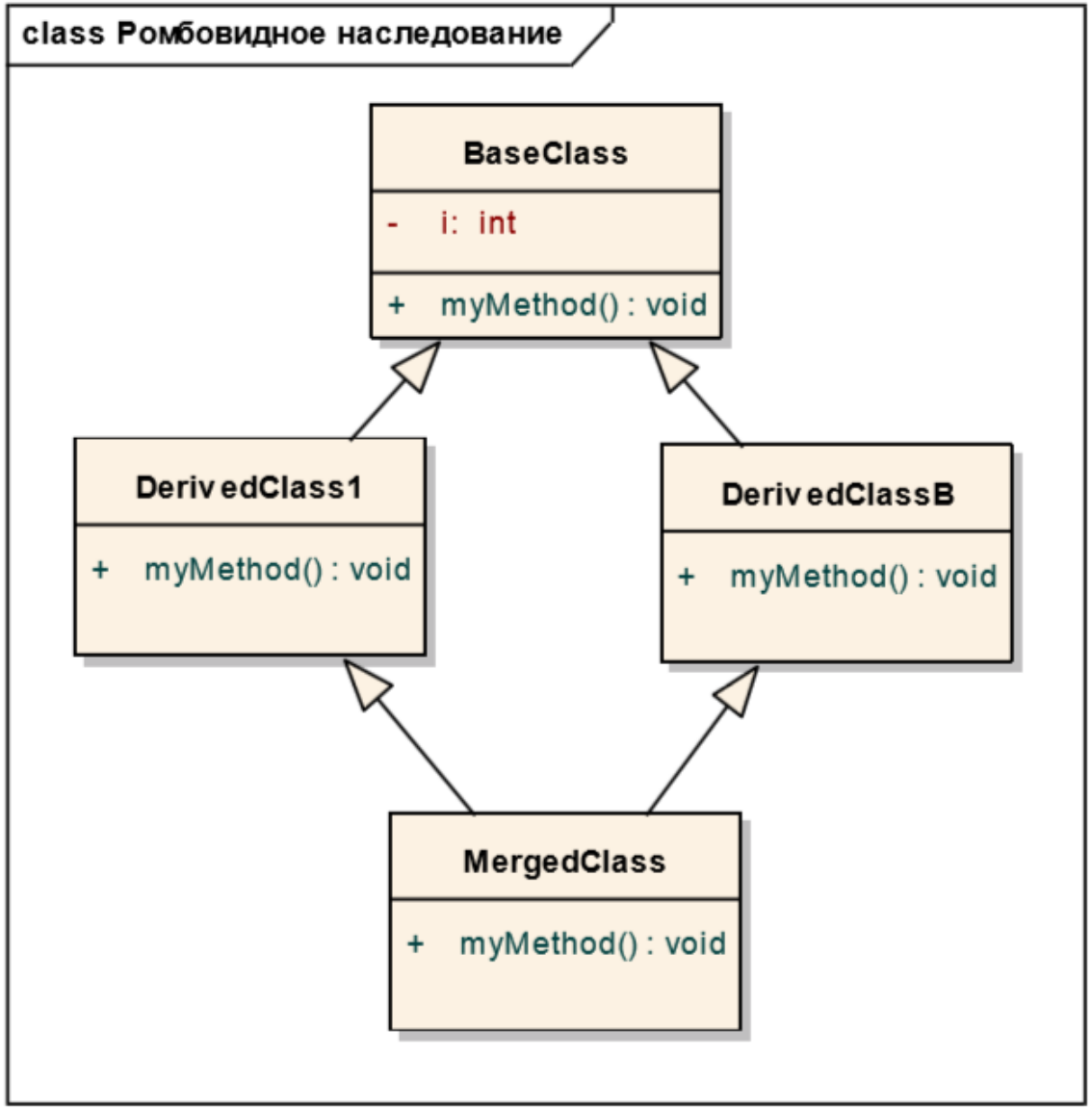
Известные проблемы:

1. Наследуются лишние поля и методы;
2. Конфликты совпадающих имен из разных веток наследования.

Множественное наследование

Одной из наиболее известных ситуаций является так называемое **ромбовидное наследование**.

1. У класса **A** имеются наследники **B** и **C;**
2. От **B** и **C** наследуется класс **D**;
3. Класс **D** получает поля и методы, имеющиеся в классе **A, в удвоенном количестве** – один комплект по линии родителя **B**, другой – по линии родителя **C**.



Интерфейсы

Интерфейсы: для описания или спецификации функциональности, которую должен реализовать каждый класс, его имплементирующий.

Т.о. **Интерфейсы –** специальная разновидность полностью абстрактных классов:

- **Не имеют** реализованных методов;
- **Не могут** включать поля данных;
- **Могут** содержать константы.

Особенности Java:

- Класс в Java должен быть наследником 1 класса-родителя.
- Класс в Java может быть наследником произвольного числа интерфейса.
- Интерфейсы также могут наследоваться от интерфейсов.

Объявление интерфейса

В качестве модификатора видимости может использоваться слово **public** (режим общей видимости).

Также модификатор может отсутствовать (пакетная видимость).

Если список предков пуст, то, в отличие от классов, интерфейс не имеет предка. При включении в интерфейс поля, оно автоматически считается общедоступным (**public**), окончательным (**final**) и переменной класса (**static**).

Сами модификаторы **public**, **static** и **final** ставить не надо. При определении в интерфейсе метода, он автоматически считается общедоступным (**public**) и абстрактным (**abstract**).

```
[<Модификатор_интерфейса>] interface <Имя_интерфейса>
[extends <Имя_интерфейса1>[, <Имя_интерфейса2>, ...,]] {
   // декларация констант;
   // декларация заголовков методов;
}
```

Основные отличия интерфейсов от классов

1. Интерфейс может включать только **методы** и **константы**.
2. Элементы интерфейса всегда имеют тип видимости **public**.
3. В интерфейсах **нет** конструкторов/деструкторов.
4. Методы не могут иметь модификаторов **abstract** (хотя абстрактные по умолчанию), **static**, **native**, **synchronized**, **final**, **private**, **protected**.
5. Интерфейс наследует все методы предка,
   но только на уровне абстракций, без реализаций.

Основные отличия интерфейсов от классов

6. Реализация интерфейса может быть только в классе.

   Если класс не абстрактный, то он должен реализовать все методы интерфейса.
7. Наследование через интерфейсы может быть множественным.
8. Наследование класса от интерфейсов может быть множественным.
9. Наследовать от нескольких интерфейсов методы с совпадающими сигнатурами, но отличными контрактами (сигнатура + тип возвращаемого значения), **нельзя**.

Композиция как альтернатива множественному наследованию

Композиция – это описание объекта:

- как состоящего из других объектов (отношение агрегации или включения как составной части)

или

- как находящегося с ними в отношении ассоциации (объединения независимых объектов).

В Java композиция любого вида – это наличие в объекте поля ссылочного типа.

Вид композиции определяется **условиями создания объекта**, который связан с этой ссылочной переменной, и **изменения этой самой ссылки.**

Композиция как альтернатива множественному наследованию

Если такой вспомогательный объект создается и уничтожается **одновременно** с главным объектом – это **агрегация**,

иначе – **ассоциация**.

Композиция – это альтернатива множественного наследования, когда наследование интерфейсов невозможно или нецелесообразно.

Но!

Что такое агрегация и ассоциация?

Композиция как альтернатива множественному наследованию

Реализация (Realization) – отношение между классификаторами (класс + интерфейс), при котором один описывает контракт (интерфейс сущности), а другой гарантирует его выполнение.

Ассоциация (Association) – объект одного класса связан с объектом другого класса и отражает некоторое между ними.

Агрегация – частный случай ассоциации, моделирующий взаимосвязь <<часть/целое>> между классами, которые в то же время могут быть равноправными, но ни один из них не является более важным, чем другой.

Пример использования композиции

Независимые классы **Car**, **Driver**, **Speed**.

Класс MovingCar наследует от Car и содержит ссылки на **Driver** и **Speed**. Особенностью объектов **MovingCar** будет то, что они включают в себя не только особенности поведения автомобиля , но и все особенности объектов типа **Driver** и **Speed**.

Автомобиль знает все о своем водителе.

Т.Е. если имеется объект **movingCar**, то **movingCar.driver** обеспечит доступ к объекту «водитель» (при условии ссылка не равной **null**),

в результате чего можно будет пользоваться общедоступными (и только!) методами этого объекта.

Аналогично строится отношение к полю **Speed**.

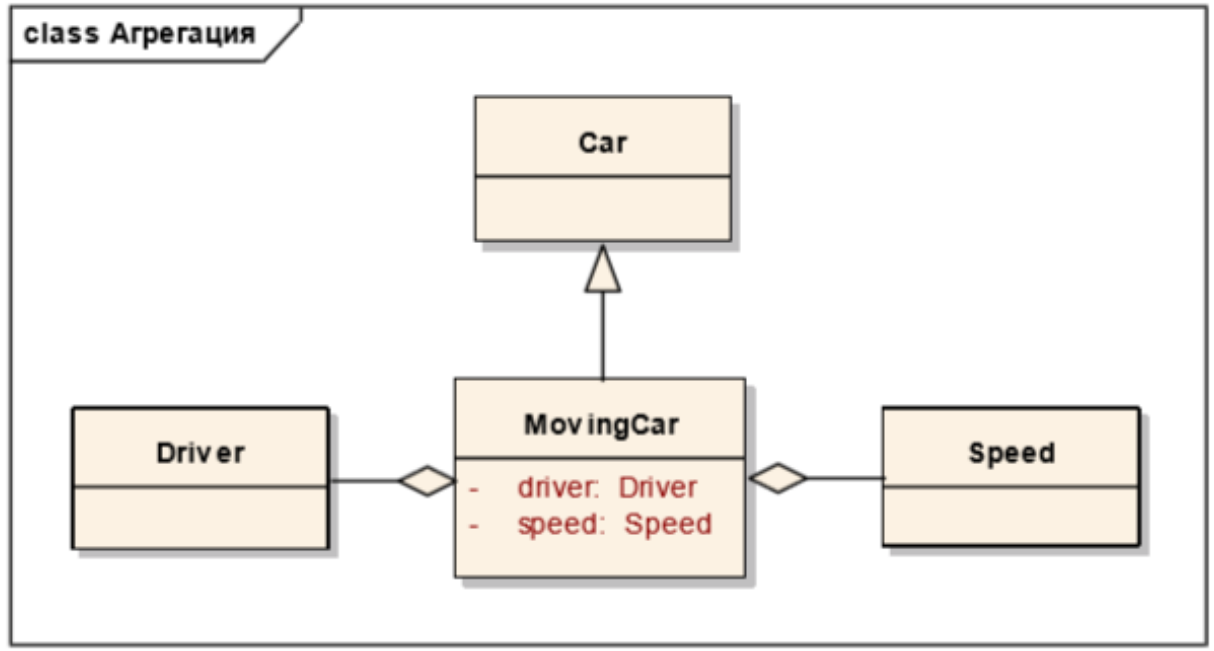
Пример использования композиции

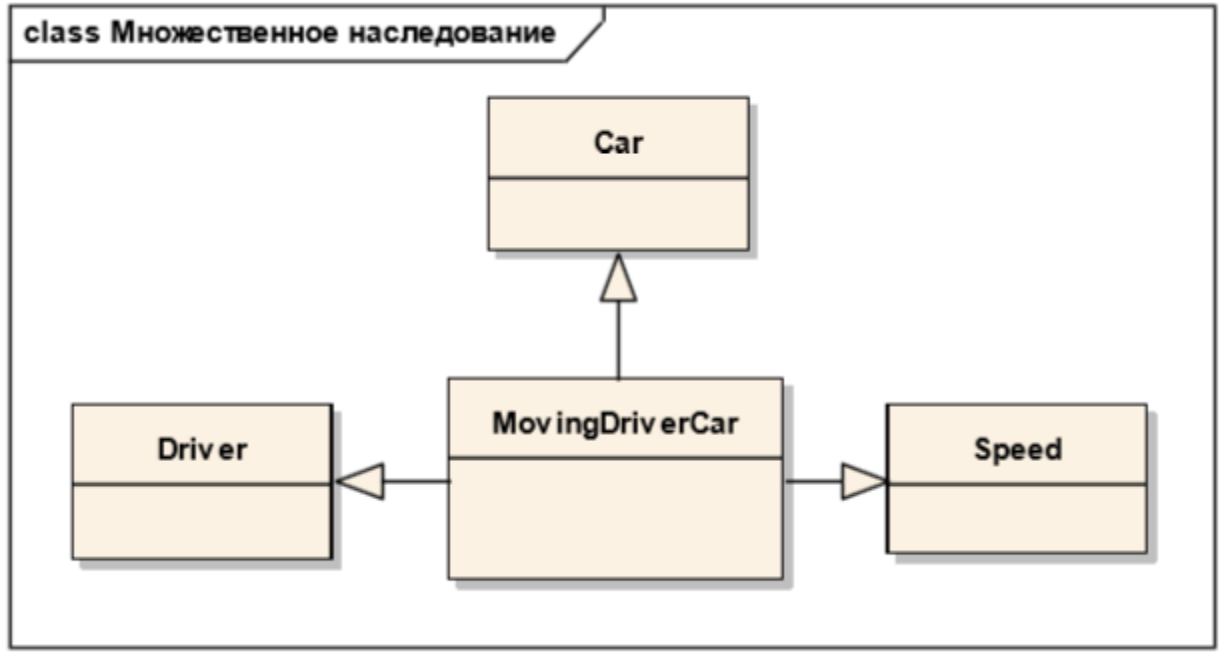
Это устраняет необходимость:

- построения гибридного класса, в котором от родителей **Car**, **Driver** и **Speed** с использованием механизма множественного наследования создается «гибрид» машино-человека, где шофера скрестили с автомобилем;
- реализации в наследнике интерфейсов, описывающих взаимодействие автомобиля с шофером и измерение/задание скорости.

Недостатки композиции:

Ограничение при использовании полиморфизма, т.к. класс-композит не является потомком своих частей.


class Агрегация
Car
MovingCar
- driver: Driver
- speed: Speed
Driver
Speed


class Множественное наследование
Car
MovingDriverCar
Driver
Speed

Встроенные классы в Java

Кроме использования наследования и ссылок, классы могут друг с другом взаимодействовать через организацию логической структуры с определением одного класса в теле другого.

ЗАЧЕМ?

Это помогает группировать классы, логически связанные друг с другом **=>** динамично управлять доступом к ним.

1. Это эффективно и понятно.
2. Один из способов сокрытия кода

Встроенные классы в Java

Почему сокрытие кода?

Т.к. внутренний класс может быть полностью недоступен и не виден вне класса-владельца.

Благодаря этому свойству внутренние классы используются как **блоки прослушивания событий.**

Типы встроенных классов

- **Вложенные (nested) классы и интерфейсы** –для задания самостоятельных классов и интерфейсов внутри классов.
  - Нестатические вложенные классы;
  - Статические вложенные классы.
- **Внутренние (inner) классы** – для создания экземпляров, принадлежащих экземплярам внешнего класса, т.е. их экземпляры не могут существовать вне объектов внешнего класса.
- **Локальные (local) классы** – задаются внутри блоков программного кода в методах или блоках инициализации (вспомогательный характер).
- **Анонимные (anonymous) классы** – совмещение декларации, реализации и вызова.; не имеют ни имени, ни конструктора и обычно используются в обработчиках событий.

Вложенные (nested) классы

- Может быть логически связан с классом владельцем,
- Но может быть использован независимо от него.

Модификатор **Static** – если такой класс заключен в интерфейс, то он становится по умолчанию **Static.**

```
class <ИмяВнешнегоКласса> {
    // тело внешнего класса
    static class <ИмяВложенногоКласса> {
        // тело вложенного класса
    }
    // продолжение тела внешнего класса
}
```

MyMath.Integral integral = new MyMath.Integral();

Доступ к полю класса осуществляется как: **<ИмяВнешнегоКласса>.<ИмяВложенногоКласса>.<имяПоля>**, а обращение к методу класса – как: **<ИмяВнешнегоКласса>.<ИмяВложенногоКласса>.<имяМетода> (<список параметров>).**

Во внешнем классе могут быть поля, которые имеют тип вложенного класса.

```java
public class Outer {
    // Статический вложенный класс
    static class Inner {
        public void show() {
            System.out.println("Метод внутреннего класса");
        }
    }

    public static void main(String[ ] args) {
        Outer.Inner inner = new Outer.Inner();
        inner.show();
    }
}
```

Вывод:

Метод внутреннего класса

Вложенные (nested) классы

1. Класс, вложенный в интерфейс, является статическим по умолчанию
2. Вложенный класс может быть базовым, производным, реализующим интерфейсы
3. Вложенный класс имеет доступ к статическим полям и методам внешнего класса
4. Подкласс вложенного класса не наследует возможность доступа к членам внешнего класса, которыми наделен суперкласс
5. Статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса
6. Статический метод вложенного класса вызывается при указании полного относительного пути к нему.

Вложенные (nested) классы

- Реализовывать интерфейс можно и в постороннем классе, т.к. имя интерфейса квалифицируется именем внешнего класса.

Задание вложенного интерфейса аналогично заданию вложенного класса:

(Вложенные интерфейсы считаются имеющими модификатор static по умолчанию).

```
class <ИмяВнешнегоКласса> {
    // тело внешнего класса
    interface <ИмяВложенногоИнтерфейса> {
        // объявление констант и заголовков методов
    }
    // продолжение тела внешнего класса
}
```

Внутренние (Inner) классы

Нестатические сложенные классы принято называть внутренними.

Доступ к элементам внутреннего класса возможен из внешнего только через объект внутреннего класса, который должен быть создан в коде метода внешнего класса.

Этот объект внутреннего класса всегда ассоциируется (**скрыто хранит ссылку**) с создавшим его объектом внешнего класса (**Enclosing** объект).

Вложенный внутренний класс может иметь любой модификатор доступа (**private**, **protected**, **package-protected**, **public**).

Аналогично для интерфейсов.

Внутренний класс **НЕ МОЖЕТ** содержать статических методов или полей.

Внутренние (Inner) классы

Внутренний класс задается аналогично вложенному, но без модификатора **static** перед именем этого класса.

Для внутренних классов экземпляры создаются через имя объекта внешнего класса, что принципиально отличает их от обычных и вложенных классов.

Созданный экземпляр внутреннего класса получает неявную ссылку на экземпляр создавшего его внешнего класса.

```
class <ИмяВнешнегоКласса> {
    // тело внешнего класса
    class <ИмяВнутреннегоКласса> {
        // тело внутреннего класса
    }
    // продолжение тела внешнего класса
}
```

```java
public class Outer {
    // Внутренний класс
    class Inner {
        public void show() {
            System.out.println("Метод внутреннего класса");
        }
    }

    public static void main(String[ ] args) {
        Outer.Inner inner = new Outer().new Inner();
        inner.show();
    }
}
```

Вывод:

Метод внутреннего класса

Ограничения на работу с внутренними классами

Пусть имеется класс **Outer**, в нем – внутренний класс **Inner** и метод **createInner()**, создающий экземпляр **Inner** и возвращающий ссылку на него. Есть класс **External**, не связанный с **Outer** и **Inner**.

Если **Inner** объявлен как **private**, то работать с его экземплярами можно только внутри конструкторов / методов **Outer**. Т.е. при размещении в теле метода класса **External** кода:

```
Outer outer = new Outer(); // Успешно
Outer.Inner inner1 = outer.new Inner(); // Ошибка
Outer.Inner inner2 = outer.createInner(); // Ошибка
```

Ограничения на работу с внутренними классами

Если **Inner** объявлен как **public**, но его конструкторы объявлены как private, то экземпляры **Inner** могут создаваться только внутри **Outer**. Если метод **Outer** вернет ссылку на экземпляр **Inner**, то к **public** полям и методам можно обращаться и из **External**:

```
Outer outer = new Outer(); // Успешно
Outer.Inner inner1 = outer.new Inner(); // Ошибка
Outer.Inner inner2 = outer.createInner(); //
Успешно
```

Если **Inner** и его конструкторы объявлены как **public**, то создавать его экземпляры можно не только в **Outer**, но и в **External**:

```
Outer outer = new Outer(); // Успешно
Outer.Inner inner1 = outer.new Inner(); // Успешно
Outer.Inner inner2 = outer.createInner(); //
Успешно
```

Внутренние (Inner) классы

1. Доступ к элементам внутреннего класса возможен только из внешнего класса через объект внутреннего.
2. Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса.
3. Объект внутреннего класса имеет ссылку на объект своего внешнего класса (Enclosing).
4. Внутренние классы не могут содержать статические поля и методы, кроме final static (константа).
5. Внутренние классы могут быть производными от других классов.
6. Внутренние классы могут быть базовыми.

Внутренние (Inner) классы

7. Внутренние классы могут реализовывать интерфейсы.
8. Внутренние классы могут быть объявлены с параметрами final, abstract, protected, public, private.
9. Если необходимо создать объект внутреннего класса вне внешнего нестатического метода класса, то необходимо определить тип объекта как <ВнешнийКласс><ВнутреннийКласс>.

Локальные классы

Никаких особенностей в применении локальных классов нет, за исключением того, что область существования их и их экземпляров ограничена тем блоком, в котором они заданы:

```
class MyApplication {
   public static void main(String[] args) {
      if (/* условие */) {
         class MyClass {
            int i;
            public MyClass(int i) { this.i = i; }
            public void print() { System.out.println(i); }
         }
         for (int i=0; i<10; i++)
            (new MyClass(i)).print();
      } else {
         MyClass obj = new MyClass(10); // Ошибка
      }
   }
}
```

```java
public class Outer {
    void outerMethod() {
        System.out.println("Метод внешнего класса");
        // Внутренний класс является локальным для метода outerMethod()
        class Inner {
            public void innerMethod() {
                System.out.println("Метод внутреннего класса");
            }
        }
        Inner inner = new Inner();
        inner.innerMethod();
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.outerMethod();
    }
}
```

Вывод:

Метод внешнего класса

Метод внутреннего класса

Локальные классы

1. Локальные внутренние классы не объявляется с помощью модификаторов доступам.
2. Внутренний класс может быть объявлен внутри метода или логического блока внешнего класса.
3. Видимость класса регулируется видимостью того блока, в котором он объявлен.

НО! Внутренний класс **сохраняет** доступ ко всем полям и методам внешнего класса, в том числе и к константам, которые были объявлены в текущем блоке кода.

Анонимные (безымянные) классы

Используются для реализации нескольких методов и создания собственных методов объекта.

Эффективно применяется только тогда, когда необходимо переопределение метода, но создавать новый класс нет необходимости.

!!! Конструктор анонимного класса определить невозможно.

Создаются 2мя путями:

1. Как наследник определяемого класса;
2. Как реализация определяемого интерфейса.

За один раз анонимный класс может или реализовать расширение класса, или реализовать интерфейс.

Не одновременно!

```
[<Имя_предка> <Имя_переменной> = ] new
<Имя_конструктора>(<Список_параметров_конструктора>) {
    // Поля и методы анонимного класса
}
```

Анонимные (безымянные) классы

```
interface MusicInstrument {
  void play();
};
...
class Musician {
  public void play(MusicInstrument i) {
    i.play();
  }
}
...
Musician maestro = new Musician();
maestro.play(new MusicInstrument() {
  public void play() { System.out.println("Piano"); }
});
maestro.play(new MusicInstrument() {
  public void play() { System.out.println("Guitar"); }
});
```

```java
public class Outer {
    // Анонимный класс, который реализует интерфейс Hello
    static Hello h = new Hello() {
        public void show() {
            System.out.println("Метод внутреннего анонимного класса");
        }
    };

    public static void main(String[ ] args) {
        h.show();
    }
}

interface Hello {
    void show();
}
```

Вывод:

Метод внутреннего анонимного класса

Анонимные (безымянные) классы

1. Расширяет другой класс / реализует интерфейс при объявлении единственного объекта. Остальным объектам будет соответствовать реализация, определяемая в самом классе.
2. Объявление анонимного класса выполняется одновременно с созданием его объектам с помощью оператор **new**.
3. Конструкторы анонимных классов **нельзя ни определить, ни переопределить!**
4. Анонимные классы допускают вложенность друг в друга.
5. Объявление анонимного класса в перечислении отличается от простого анонимного класса, т.к. инициализация всех элементов происходит при первом обращении к нему.

На сегодня все!
Увидимся на следующей неделе!

На следующей лекции мы поговорим

о базовых объектах Java.