# АГРЕГИРОВАНИЕ И НАСЛЕДОВАНИЕ

**Цели**

Цель этой лабораторной работы — познакомить студентов с понятиями агрегирования и наследования классов.

Отслеживаемые аспекты:

- изучение механизма наследования
- понимание разницы между наследованием и агрегацией
- downcasting и upcasting

## Агрегация и композиция

Агрегация и композиция относятся к наличию ссылки на объект в другом классе. Этот класс в основном будет повторно использовать код из соответствующего класса объекта. Пример:

Композиция:

```
public class Foo {
   private Bar bar = new Bar();
}
```

Агрегация:

```
public class Foo {
   private Bar bar;

   // Объект Bar может продолжать существовать, даже если объект Foo не
   //существует.
   Foo(Bar bar) {
      this.bar = bar;
   }
}
```

Практический пример:

```
class Page {
   private String content;
   public int numberOfPages;

   public Page(String content, int numberOfPages) {
      this.content       = content;
      this.numberOfPages  = numberOfPages;
   }
}

class Book {
   private String title;                          // Композиция
   private Page[] pages;                          // Композиция
   private LibraryRow libraryRow = null;          // Агрегация

   public Book(int size, String title, LibraryRow libraryRow) {
       this.libraryRow = libraryRow;
       this.title = title;
```

```
        pages = new Page[size];

        for (int i = 0; i < size; i++) {
          pages[i] = new Page("Page " + i, i);
        }
    }
}

class LibraryRow {
    private String rowName = null;                    // Агрегация

    public LibraryRow(String rowName) {
        this.rowName = rowName;
    }
}

class Library {

    public static void main(String[] args) {
        LibraryRow row = new LibraryRow("a1");
        Book book = new Book(100, "title", row);

        // После того, как ссылка на объект Book больше не существует,
        // Garbage Collector удалит его (в какой-то момент, не обязательно сразу)
       // удалит этот экземпляр, но объект LibraryRow, переданный конструктору,
       // не будет затронут.

        book = null;
    }
}
```

- **Агрегация** (aggregation) – объект-контейнер может существовать и при отсутствии агрегированных объектов, поэтому он считается слабой ассоциацией (*weak association*). В приведенном выше примере книжная полка может существовать без книг.
- **Композиция** (composition) – это сильная агрегация, указывающая на то, что существование объекта зависит от другого объекта. При исчезновении объектов, содержащихся в композиции, существование объекта-контейнера прекращается. В приведенном выше примере книга не может существовать без страниц.

**Инициализация** содержащихся объектов может выполняться в 3 различных момента времени:

- при **определении** объекта (до конструктора: с использованием либо начального значения, либо блоков инициализации);
- внутри **конструктора**;
- непосредственно **перед использованием** (этот механизм называется ленивой инициализацией (*lazy initialization*)).

## Наследование (Inheritance)

**Наследование**, также называемое деривацией, представляет собой механизм повторного использования кода, специфичный для объектно-ориентированных языков, и представляет собой возможность определения класса, расширяющего другой уже

существующий класс. Основная идея состоит в том, чтобы взять существующую функциональность в классе и добавить новую или смоделировать существующую.

Существующий класс называется **родительским классом**, **базовым классом** или **суперклассом**. Класс, расширяющий родительский класс, называется **дочерним классом (child)**, **производным классом** или **подклассом**.

В отличие от C++, Java не допускает *множественного наследования* (*multiple inheritance*), поэтому мы не можем столкнуться с двусмысленностями, подобными проблеме ромба (Diamond Problem). Всякий раз, когда мы хотим обратиться к родительскому методу (используя ключевое слово super, как мы увидим ниже), этот родительский метод определяется однозначно.

## Агрегация vs. Наследование

***Когда использовать наследование, а когда агрегацию?***

Ответ на этот вопрос во многом зависит не только от данных анализируемой задачи, но и от замысла проектировщика, общедействительного рецепта на этот счет не существует. В общем, **агрегация** используется, когда кто-то хочет использовать возможности одного класса внутри другого класса, но не его интерфейс (посредством наследования новый класс также будет предоставлять методы базового класса). Мы можем выделить два случая:

- иногда желательно реализовать функциональность объекта, содержащегося в новом классе, и **ограничить** действия пользователя только методами нового класса (точнее, желательно не позволять пользователю использовать методы старого класса)). Для достижения этого эффекта в новый класс будет добавлен объект типа содержащегося класса, имеющий спецификатор частного доступа.
- содержащийся объект (агрегат) должен/хотеть иметь **прямой** доступ. В этом случае мы будем использовать спецификатор публичного доступа. Примером этого может быть класс Car, который содержит в качестве общедоступных членов объекты типа Engine, Wheel и т. д.

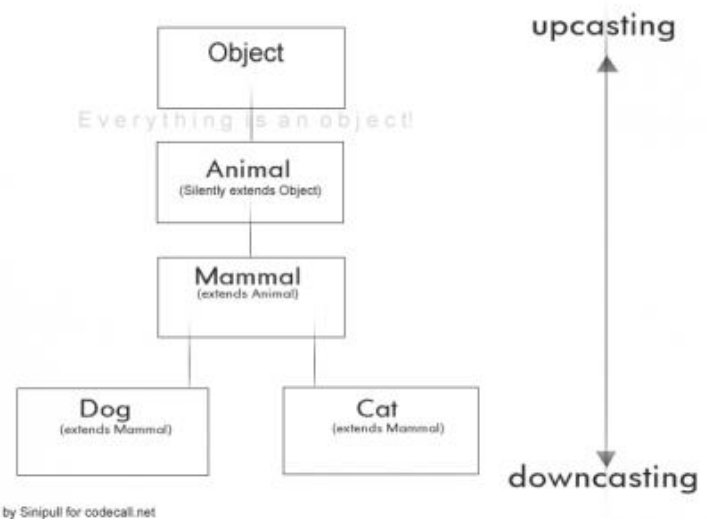
**Наследование** — это механизм, позволяющий создавать «специализированные» версии существующих (базовых) классов. Наследование обычно используется, когда нужно создать тип данных, представляющий конкретную реализацию (специализацию, предоставляемую производным классом) чего-то более общего. Простым примером может служить класс Dacia, который наследует класс Car.

**Разница** между наследованием и агрегацией на самом деле представляет собой разницу между двумя основными типами отношений, существующих между объектами приложения:

- **is a** - указывает, что класс является производным от базового класса (интуитивно, если у нас есть класс Animal и класс Dog, то было бы нормально, чтобы Dog был производным от Animal, другими словами, Dog *is an* Animal);
- **has a** - указывает, что класс-контейнер содержит класс, содержащийся в нем (интуитивно, если у нас есть класс Car и класс Engine, то было бы нормально иметь ссылку на Engine внутри Car, другими словами, Car *has an* Engine)

## Upcasting и Downcasting

**Преобразование** ссылки на производный класс в ссылку на базовый класс называется повышающим преобразованием **(upcasting)**. Преобразование **(Upcasting)** выполняется автоматически и не требует явного объявления программистом.




Пример апкастинга (upcasting):

```
class Instrument {
   public void play() {}

   static void tune(Instrument i) {
      i.play();
   }
}

// Объекты Wind — это инструменты
// потому что у них одинаковый интерфейс:
public class Wind extends Instrument {
   public static void main(String[] args) {
      Wind flute = new Wind();
      Instrument.tune(flute); // !! Upcasting автоматически, поскольку метод получает
                              // объект Instrument, а не объект Wind. Поэтому было бы
                              //излишним явное приведение типов, например:
                              // Instrument.tune((Instrument) flute)
   }
}
```

Хотя объект флейты (**flute**) является экземпляром класса **Wind**, он передается в качестве параметра вместо объекта **Instrument**, который является суперклассом класса **Wind**. Преобразование выполняется при передаче параметра. Термин «восходящее приведение» (**upcasting**) происходит от диаграмм классов (особенно UML), где наследование представлено двумя блоками, расположенными один под другим и представляющими два класса (верхний — базовый класс, нижний — производный класс), соединенные стрелкой, указывающей на класс. базовый.

**Понижающее приведение (downcasting )** — это операция, **обратная** восходящему приведению (**upcasting**), и представляет собой явное преобразование типов, спускающееся по иерархии классов (преобразование базового класса в производный). **Это приведение должно быть выполнено программистом явно**. Понижающее приведение возможно только в том случае, если объект, объявленный принадлежащим базовому классу, на самом деле является экземпляром пониженного производного класса.

Вот пример использования понижающего приведения (downcasting):

```
class Animal {
   public void eat() {
      System.out.println("Animal eating");
   }
}

class Wolf extends Animal {
   public void howl() {
      System.out.println("Wolf howling");
   }

   public void eat() {
      System.out.println("Wolf eating");
   }
}

class Snake extends Animal {
   public void bite() {
      System.out.println("Snake biting");
   }
}

class Test {
   public static void main(String[] args) {
      Animal a [] = new Animal[2];

      a[0] = new Wolf();    // Upcasting automat
      a[1] = new Snake();   // Upcasting automat

      for (int i = 0; i < a.length; i++) {
         a[i].eat(); // 1

         if (a[i] instanceof Wolf) {
            ((Wolf)a[i]).howl(); // 2
         }

         if (a[i] instanceof Snake) {
            ((Snake)a[i]).bite(); // 3
         }
      }
   }
}
```

Результаты:

```
Wolf eating
Wolf howling
Animal eating
Snake biting
```

Строки, отмеченные цифрами **2** и **3**, выполняют преобразование от Animal к Wolf и Snake соответственно, чтобы можно было вызвать определенные методы, определенные в этих

классах. Перед выполнением приведения типа вниз (преобразование типа в Wolf или Snake) мы проверяем, имеет ли соответствующий объект желаемый тип (с помощью оператора **instanceof**). Если бы мы попытались преобразовать экземпляр объекта, созданного в Snake, в тип Wolf, виртуальная машина сигнализировала бы об этом, выдав исключение при запуске программы.

Вызов метода eat() (**строка 1**) осуществляется напрямую, без приведения вниз, поскольку этот метод также определен в базовом классе Animal. Поскольку Wolf переопределяет метод eat(), вызов a[0].eat() отобразит «Wolf eat». Вызов a[1].eat() приведет к вызову метода базового класса (выходные данные отобразят «Animal eating»), поскольку экземпляр a[1] создается для Snake, и Snake не переопределяет метод eat().

Апкастинг — очень важный элемент. Часто ответ на вопрос: *есть ли необходимость в наследстве*? даёт ответ на вопрос: *нужен ли мне апкастинг*? Это связано с тем, что преобразование выполняется, когда один или несколько объектов из производных классов выполняют тот же метод, который определен в родительском классе.

**Давайте попробуем избежать использования instanceof**

Однако, несмотря на то, что мы проиллюстрировали, как instanceof может помочь нам понять, к чему следует приводить, предпочтительнее организовать наши классы и дизайн кода таким образом, чтобы позволить языку Java автоматически выполнять проверку типов и вызывать соответствующий метод. Мы проведем рефакторинг предыдущего кода, чтобы не нужен экземпляр instanceof:

```
class Animal {
   public void eat() {
      System.out.println("Animal eating");
   }

   public void action() {
      // нам нужен этот метод, потому что мы создадим вектор с экземплярами Animal
     // и вызовем для них этот метод
   }
}

class Wolf extends Animal {
   public void action() {
      System.out.println("Wolf howling");
   }

   public void eat() {
      System.out.println("Wolf eating");
   }
}

class Snake extends Animal {
   public void action() {
      System.out.println("Snake biting");
   }
}

class Test {
   public static void main(String[] args) {
      Animal a [] = new Animal[2];

      a[0] = new Wolf();
```

```
        a[1] = new Snake();

        for (int i = 0; i < a.length; i++) {
            a[i].eat();

            // теперь, когда они называются одинаково, мы можем вызвать метод action
            // класса Animal (обратите внимание, почему нам нужно было определить
            // метод действия в классе Animal), и соответствующий метод будет вызван
            // для конкретного типа экземпляра a[i ]

            a[i].action();
        }
    }
}
```

Результаты:

```
Wolf eating
Wolf howling
Animal eating
Snake biting
```

**Последствия наследования**

В Java классы и их члены (методы, переменные, внутренние классы) могут иметь различные спецификаторы доступа, показанные в вики по организации ресурсов и контролю доступа.

- спецификатор доступа **protected** – указывает, что к соответствующему члену или методу можно получить доступ только изнутри самого класса или из классов, производных от этого класса. Классы не могут иметь этот спецификатор, только их члены!
- спецификатор доступа **private** – указывает, что к соответствующему члену или методу можно получить доступ только изнутри самого класса, а не из классов, производных от этого класса. Классы не могут иметь этот спецификатор, только их члены!

Конструкторы **не** наследуются и могут вызываться только в контексте дочернего конструктора. Вызовы конструктора объединены в цепочку, что означает, что перед инициализацией дочернего объекта сначала будет инициализирован родительский объект. Если родительский элемент в свою очередь является дочерним, его родительский элемент будет инициализирован (пока не достигнет высшего родительского элемента – root).

Помимо повторного использования кода, наследование дает возможность разрабатывать приложение шаг за шагом (этот процесс называется *инкрементальной разработкой*). Таким образом, мы можем использовать уже работающий код и добавлять к нему новый код, изолируя таким образом ошибки во вновь добавленном коде. Для получения дополнительной информации прочтите главу «*Reusing Classes*» в книге «*Thinking în Java*» (Брюс Экель).

**Переопределение, перегрузка и скрытие статических методов**

**Переопределение** (*overriding*) включает замену функциональности родительского класса(ов) для текущего экземпляра. **Перегрузка** (*overloading*) предполагает предоставление дополнительной функциональности либо для методов текущего класса, либо для родительского класса(ов).

```
public class Car {
    public void print() {
        System.out.println("Car");
    }
```

```
    public void init() {
        System.out.println("Car");
    }

    public void addGasoline() {
        // do something
    }
}

class Dacia extends Car {
    public void print() {
        System.out.println("Dacia");
    }

    public void init() {
        System.out.println("Dacia");
    }

    public void addGasoline(Integer gallons) {
        // do something
    }

    public void addGasoline(Double gallons) {
        // do something
    }
}
```

Зависящие от экземпляра методы являются полиморфными (во время выполнения они могут иметь разные реализации), поэтому их можно *переопределить* или *перегрузить*. Метод **print** переопределяется в классе Dacia, а это означает, что в любом экземпляре, даже если он приведен к типу Car, вызываемым методом всегда будет метод **print** из класса Dacia. Метод **addGasoline** перегружен, что означает, что мы можем выполнять методы с разными сигнатурами, но с одинаковым именем (наиболее часто используемым при создании методов преобразования).

```
Car a = new Car();
Car b = new Dacia();
Dacia c = new Dacia();
Car d = null;

a.print(); // покажет Car
b.print(); // покажет Dacia
c.print(); // покажет Dacia
d.print(); // выдает исключение NullPointerException
```

Переопределение также не применяется к статическим методам, поскольку они не зависят от экземпляра. Если в приведенном выше примере мы сделаем методы **print** из Car и из Dacia статическими, результат будет следующим:

```
Car a = new Car();
Car b = new Dacia();
Dacia c = new Dacia();
```

```
Car d = null;

a.print(); // покажет Car
b.print(); // покажет Car потому что инициализированный тип b равен Car
c.print(); // покажет Dacia потому что инициализированный тип c равен  Dacia
d.print(): // покажет Car потому что инициализированный тип d равен Car
```

Синтаксис Java позволяет вызывать статические методы для экземпляров (например, a.print вместо Car.print), но это считается плохой практикой, поскольку может усложнить понимание кода.

**Правильное переопределение метода equals(Object o)**

Одной из наиболее распространенных проблем является правильное переопределение метода *equals*. Ниже мы можем увидеть пример некорректного переопределения этого метода.

```
public class Car {
   public boolean equals(Car c) {
      System.out.println("Car");
      return true;
   }

   public boolean equals(Object o) {
      System.out.println("Object");
      return false;
   }
}
```

Первый метод является **перегрузкой** метода equals, а второй метод – **переопределением** метода equals.

```
Car a = new Car();
Dacia b = new Dacia();
Int c = new Int(10);

a.equals(a); // покажет Car
a.equals(b); // покажет Car потому что апкастинг делается с Dacia на Car
a.equals(c); // покажет Object потому что апкастинг делается с Int на Object
```

Проблема, которую можно наблюдать, заключается в том, что мы можем передавать в качестве аргументов методу **equals** типы данных, отличные от Car, что может привести к возникновению исключений приведения или когда мы хотим получить доступ к определенным свойствам из экземпляра. Ниже приведен правильный способ переопределить метод **equals**.

```
public class Car {
   public boolean equals(Car c)
   {
      return true;
   }

   public boolean equals(Object o)
   {
      if (o == this) {
         return true;
```

```
        }

        if (!(o instanceof Car)) {
            return false;
        }

        return equals((Car) o);
    }
}
```

Обратите внимание, что использование instanceof не рекомендуется, но в данном случае это единственный способ гарантировать, что экземпляр объекта, отправленный в метод, имеет тип Car.

**Ключевое слово супер. Использование**

Ключевое слово super относится к родительскому экземпляру текущего класса. Его можно использовать двумя способами: вызвать переопределенный (*overriden*) метод или вызвать родительский конструктор.

**Вызов переопределенного метода**

```
public class Superclass {

    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}

public class Subclass extends Superclass {

    // overrides printMethod in Superclass
    public void printMethod() {
        super.printMethod();  // вызов родительского метода

        System.out.println("Printed in Subclass.");
    }

    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Результаты:

```
Printed in Superclass.
Printed in Subclass.
```

**Вызов родительского конструктора**

```
class Superclass {
    public Superclass() {
      System.out.println("Printed in Superclass constructor with no args.");
    }

    public Superclass(int a) {
```

```
        System.out.println("Printed in Superclass constructor with one integer argument.");
    }
}

class Subclass extends Superclass {
    public Subclass() {
        super();    // вызывает родительский конструктор
                    // этот вызов должен находиться в первой строке конструктора!!

        System.out.println("Printed in Subclass constructor with no args.");
    }

    public Subclass(int a) {
        super(a);   // вызывает родительский конструктор
                    // этот вызов должен находиться в первой строке конструктора!!

        System.out.println("Printed in Subclass constructor with one integer argument.");
    }

    public static void main(String[] args) {
        Subclass s1 = new Subclass(20);
        Subclass s2 = new Subclass();
    }
}
```

Результаты:

```
Printed in Superclass constructor with one integer argument.
Printed in Subclass constructor with one integer argument.
Printed in Superclass constructor with no args.
Printed in Subclass constructor with no args.
```

Вызов родительского конструктора **должен** быть первой строкой в конструкторе подкласса, если родительский вызов существует (мы вполне можем не вызывать **super** из конструктора).
Даже если вызов метода **super**() не указан, компилятор автоматически вызовет родительский конструктор по умолчанию, но если требуется другой конструктор, соответствующий вызов **super(args)** является обязательным.

**Резюме**

**Отношения между объектами:**

- Агрегация - **has a**
- Наследование - **is a**

**Upcasting**

- преобразование **дочерний элемент ⇒ родительский**
- делается автоматически

**Downcasting**

- преобразование **родительский элемент ⇒ дочерний**
- должно быть сделано программистом явно
- старайтесь избегать использования оператора **instanceof**

**Переопределение**

- замена функциональности метода базового класса в производном классе
- сохраняет имя и подпись метода

**Перегрузка**

- внутри класса может быть несколько методов с одним и тем же именем при условии, что сигнатура (тип, аргументы) различна

**super**

- экземпляр родительского класса
- помните из предыдущей лабораторной работы, что **this** относится к текущему экземпляру класса

**Упражнения**

Gigel хочет сделать подарок своей матери на день рождения и знает, что она очень любит конфеты. Ему нужна ваша помощь, чтобы сделать самый красивый и вкусный подарок:

**Task 1** [2p]

Вы разработаете класс CandyBox, который будет содержать частные (private) поля flavor (String) и origin (String). В классе также будет:

- конструктор без параметров
- конструктор, который инициализирует все поля
- метод типа float getVolume(), который будет возвращать значение 0;
- Поскольку класс Object находится в корне дерева наследования любого класса, любой экземпляр будет иметь доступ к ряду возможностей, предоставляемых Object. Одним из них является метод toString(), целью которого является предоставление представления экземпляра в виде строки символов, используемой при вызове System.out.println(). Добавьте метод toString(), который будет возвращать вкус и регион происхождения коробки конфет.

**Task** 2 [2p]

От него происходят классы *Lindt, Baravelli, ChocAmor*. Для интересного дизайна коробки будут иметь разную форму:

- *Lindt* будет содержать длину (length), ширину (width) и высоту (height ) (float);
- *Baravelli* будет цилиндром. Он будет содержать поле рдиуса (radius ) и высоты (height) (float);
- *ChocAmor*, будучи кубом, будет содержать длину (length) (float);

Классы будут иметь:

- конструкторы без параметров
- конструкторы, позволяющие инициализировать члены. Определите способ повторного использования существующего кода. Для каждого типа коробки вы инициализируете в конструкторе поля вкуса (flavor) и происхождения (origin) с соответствующим типом.
- Переопределить метод getVolume(), чтобы он возвращал конкретный объем каждой коробки конфет в зависимости от ее типа.
- Переопределить метод toString() в производном классе, чтобы он использовал реализацию метода toString() в базовом классе. Вернуть сообщение вида *“The ” + origin + “ ” + flavor + “ has volume ” + volume*;

**Task 3** [1p]

Добавьте метод **equals**() в класс CandyBox. Обоснуйте выбранный критерий эквивалентности. См. методы класса Object, унаследованные всеми классами. В Object есть метод **equals**, реализация которого проверяет равенство объектов путем сравнения ссылок. **Hint:** Вы можете автоматически сгенерировать метод, используя IDE. Выберите рассматриваемые поля и проанализируйте, как будет переопределен метод **equals**.

**Task 4** - *Upcasting* [2p]

Теперь, когда мы определили тип коробок конфет, мы можем собрать подарок, оставив на ваше усмотрение, каким будет его дизайн. В пакете java.util есть класс ArrayList, определяющий массив изменяемого размера со специфическими методами (add, size, get, их полный список есть в документации). Создайте класс CandyBag, который будет содержать ArrayList из нескольких коробок каждого типа. Создайте объекты Lindt и проверьте их равенство.

**Task 5** - *Downcasting* [1p]

Добавьте в класс Baravelli функцию printBaravelliDim(), которая будет отображать размеры радиуса и высоты. Аналогично поступаем с остальными типами коробок, добавляя методы printChocAmorDim() и printLindtDim(), в которых нужно отображать размеры каждой коробки.

**Task 6** - *Agregare* [2p]

Gigel захочет отправить подарок с курьером, чтобы мама не нашла его раньше. Помогите ему определить местоположение, создав класс «Area», который будет содержать объект CandyBag, поле «number» (int) и поле «street» (String). Класс также будет иметь:

- конструктор без параметров
- конструктор, который инициализирует все поля
- Теперь, когда мы завершили сборку, мы сообщим маме о ее подарке с помощью открытки. Создайте метод getBirthdayCard(), который сначала будет отображать полный адрес, а затем сообщение с днем рождения.
- Также здесь пройдитесь по массиву, вызвав метод toString() для его элементов.
- Пройдитесь по массиву и, используя приведение к соответствующему классу, вызовите методы, специфичные для каждого класса. Чтобы определить тип текущего объекта, используйте оператор **instanceof**.
- Наконец, измените подход к предыдущей программе, чтобы вам больше не требовался **instanceof**.