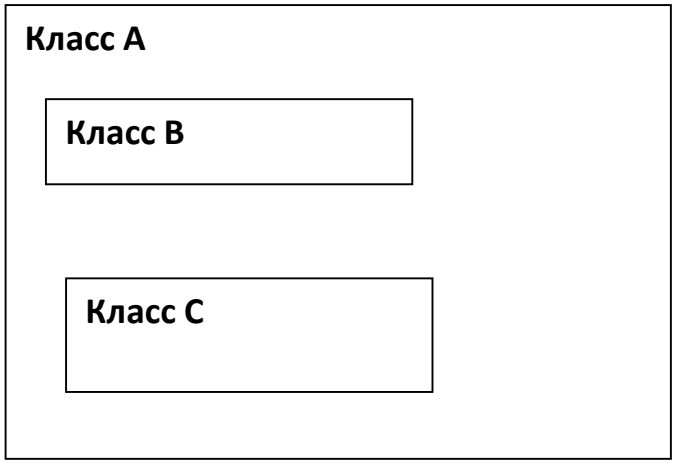
Способы построения сложных объектов

- Вложенные классы
- Включение объектов
- Наследование классов

Вложенные классы

Вложенные классы — это классы (B и C) для выделения в программе некой сущности, которая неразрывно связана с другой сущностью (A). Вложенный класс создают, чтобы скрыть его переменные и методы от внешнего мира.



- Это хороший способ группировки классов, которые используются только в одном месте.
- Для инкапсуляции.
- Улучшение читаемости кода

Типы вложенных классов

```
class Base {
      void method1() {   }
      void method2() {   }
    }
    class A { // нормальный класс
      static class B {   } // статический вложенный класс
      class C {   } // не статический вложенный класс
      void f() {
        class D {     } // локальный внутренний класс (внутри метода)
      }

      void g() {
        Base bref = new Base() { // анонимный внутренний класс
//  Анонимный класс является подклассом существующего класса Base.
          void method1() {    }
        };
      }
    }
```

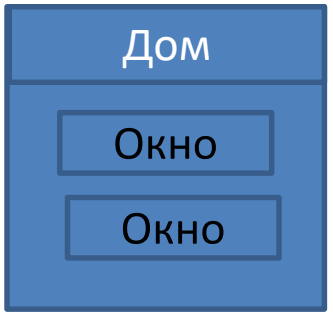
Пример нестатического вложенного (внутреннего) класса

Внутренний класс имеет доступ ко всем полям и методам обрамляющего класса

```
public class OuterClass { // внешний класс
        public void method() { ... }

        public class InnerClass { // внутренний класс
            public InnerClass () { … }
              public void method() { ... }

           public void anotherMethod() {
              method(); // вызов method InnerClass
             OuterClass.this.method() // вызов method OuterClass
           }
        }
    }
```



Создание экземпляра вложенного класса, объект внутреннего класса не может существовать без объекта «внешнего» класса.

```
OuterClass  oclass = new  OuterClass(); // создание объекта внешнего класса
OuterClass. InnerClass  iclass = oclass.new  InnerClass();  // создание внутреннего объекта
```

Нестатические вложенные классы не могут иметь static поля, исключение составляют константы, объявленные как static final.

Вложенные статические классы

Используются для создания объектов независимых от внешнего класса, т.е. когда нет явной связи между ними

```
class OuterClass {
   ...
   static class StaticNestedClass { // статический вложенный класс
      ...
   }
   class InnerClass { // не статический вложенный класс
      ...
   }
}
```

Объект не статического вложенного класса можно создать только как экземпляр внешнего класса

InnerClass MyIC= new OuterClass().new InnerClass();

Объект статического вложенного класса можно создавать отдельно (привязан к классу, а не объекту)

StaticNestedClass MySNC = new OuterClass. StaticNestedClass();

Объект статического вложенного класса может существовать сам по себе. Слово static в объявлении внутреннего класса не означает, что можно создать всего один объект.

Пример статического вложенного класса

Объект статического класса не хранит ссылку на конкретный экземпляр внешнего класса, поэтому статические вложенные классы, не имеют доступа к нестатическим полям и методам обрамляющего класса

```
class Outer3 {
String name;
. . .
static class Inner3 {
. . .
public void f(Outer3 obj) { System.out.println(obj.name); // Здесь без obj нельзя
} }
. . .
public static Inner3 createInner() { return new Inner3(); }
. . .
}
Outer3.Inner3 obj1 = new Outer3.Inner3(); // явное порождение
// порождение через метод createInner()
Outer3.Inner3 obj2 = Outer3.createInner();
```

Локальные внутренние классы

Локальные внутренние классы определяются в блоке Java кода. Основное применение локальные классы находят в тех случаях, когда необходимо написать класс, который будет использоваться внутри одного метода.

У локальных внутренних классов следующие ограничения:

- объекты локального класса могут создаваться только в блоке кода, котором они описаны.
- объект локального класса, объявленный внутри блока кода другого класса, не является членом класса, к которому относится блок;
- объекты локального класса видны только в пределах блока, в котором объявлены;
- локальные внутренние классы не могут быть объявлены как **private, public, protected или static** (применимы только к членам класса);
- они не могут иметь внутри себя статических объявлений (полей, методов, классов); исключением являются константы (static final);
- в Java 7 локальный класс может получить доступ к локальной переменной или параметру метода, только если они объявлены в методе как final (в Java 8 можно обращаться из локального класса не только к финальным переменным)
- из локального внутреннего класса можно получить доступ ко всем членам внешнего класса.

```
public class Handler {
        public void handle(final String requestPath) {
           class LocalClass {
                              LocalClass () {…};
                              . . .
                              }
              LocalClass lc = new LocalClass();
                              . . .
}
```

Анонимные классы

Анонимный класс - это локальный класс без имени. Анонимный класс является подклассом существующего класса или реализацией интерфейса.

Использование анонимных классов :

- тело класса является очень коротким;
- нужен только один экземпляр класса;
- отсутствует конструктор;
- класс используется в месте его создания или сразу после него;
- имя класса не важно и не облегчает понимание кода.

Объявление анонимного класса

```
new <имя класса или интерфейса>() {
        // описание данных и методов анонимного внутреннего класса
}
class Button {
   public void click() {
      System.out.println("Нажатие на кнопку");
   }
}
 Button btnCopy = new Button() {
         public void click() {
            System.out.println("Копирование данных");
         }
      };
```

Включение объектов

Включение (один объект является составной частью другого объекта или объект ссылается на другой объект), реализует отношение «является частью» («HAS A»).

Композиция

Один объект создается вместе с другим объектом и время жизни "части" зависит от времени жизни целого (жесткая связь)

```
class Окно {….}

class Дом {
 private  Окно  _o  =
new Окно();
…….
  }
```

Агрегация

Объекты создаются независимо и один объект получает ссылку на другой объект в процессе конструирования (не строгая связь),

```
class Преподаватель {….}
class Кафедра{
private  Преподаватель  _p;
    public Кафедра(
Преподаватель  y)
   {  _p = y; }
……….
   }
Преподаватель p=new
Преподаватель();
Кафедра k=new Кафедра(p);
```

Ассоциация

Объекты двух классов могут ссылаться один на другой, эта связь устанавливается не в процессе создания объекта (не строгая связь)

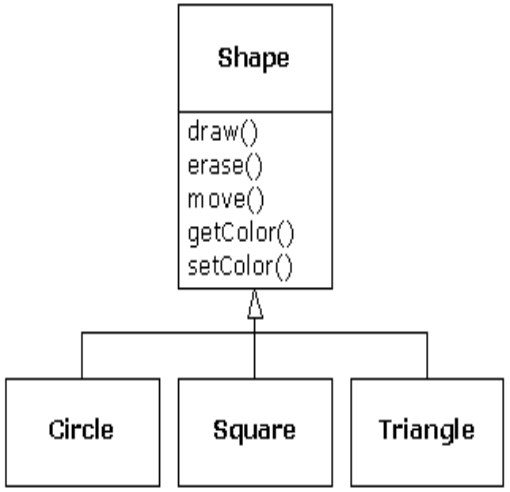
```
public class Программист {
private Компьютер computer;
public Программист() {}
…….
}
public class Компьютер {
private Программист
programmer;
public Компьютер() {}
…….
}
```

Наследование классов

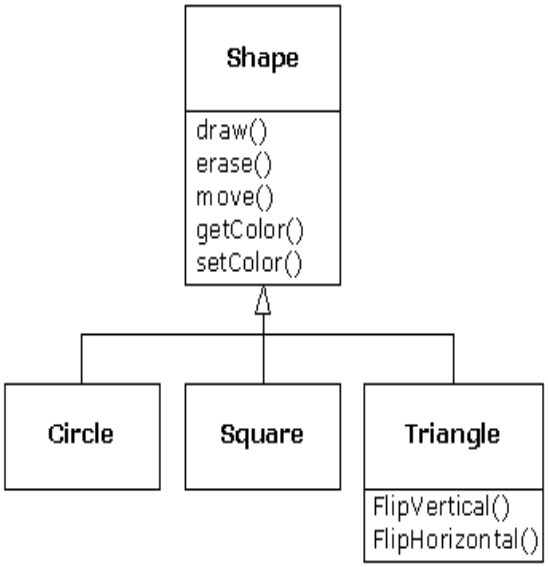
Механизм наследования поддерживает концепцию иерархии классов, объект является разновидностью другого объекта, реализует отношение «является»(IS A) .

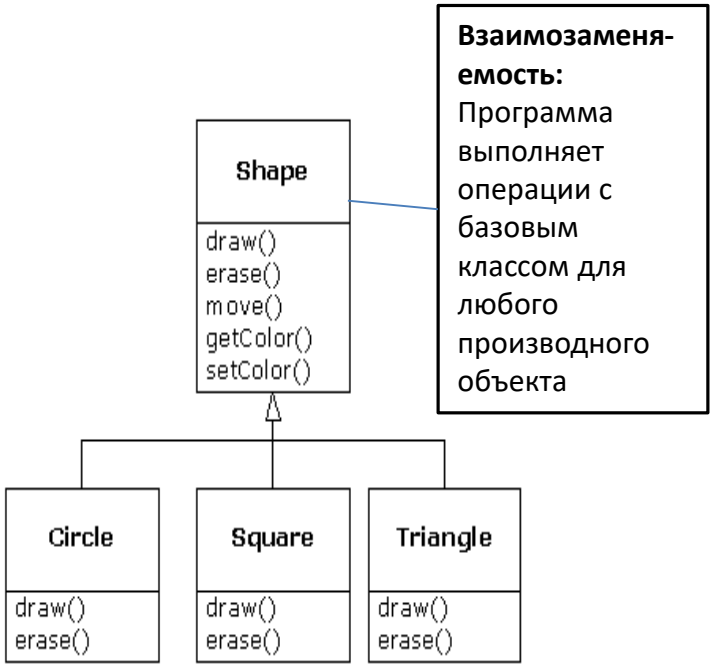
Наследование используются в следующих случаях:

1. Расширение базового класса путем добавления новых данных и методов (повторное использование кода).
2. Изменение поведения функций базового класса (перегрузка функций).
3. Взаимозаменяемые объекты (полиморфизм и позднее связывание).

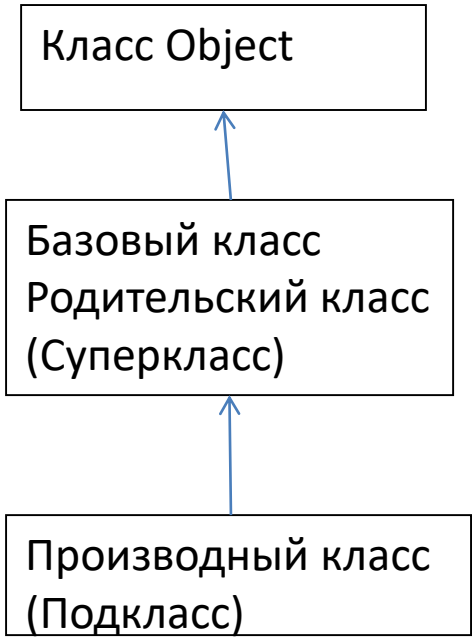


Наследование подразумевает, что не private компоненты и методы базового класса доступны в производном классе





Спецификация базового класса




```
class A extends B
{
.....
}
```

Суперкласс указывается ключевым словом extends. Он должен быть доступным классом и не иметь модификатора final.

Классы, для которых не указан расширяемый класс, являются неявным расширением класса Object.

Object oref = new B();

Описание класса Object

public class Object {

Object();

Object clone(); Функция создания нового объекта, копии существующего

boolean equals(Object object); Функция определения равенства текущего объекта другому

void finalize(); Процедура завершения работы объекта; вызывается перед удалением неиспользуемого объекта

Class<?> getClass(); Функция определения класса объекта во время выполнения

hashCode(); Функция получения хэш-кода объекта

void notify(); Процедура возобновления выполнения потока, который ожидает на мониторе этого объекта.

void notifyAll(); Процедура возобновления выполнения всех потоков, которые ожидают на мониторе этого объекта

String toString(); Функция возвращает строку описания объекта (package.class@hashCode)

void wait(); Остановка текущего потока до тех пор, пока другой поток не вызовет notify() или notifyAll метод для этого объекта

void wait(long ms); Остановка текущего потока на время или до тех пор, пока другой поток не вызовет notify() или notifyAll метод для этого объекта

void wait(long ms, int nano); Остановка текущего потока на время или до тех пор, пока другой поток не вызовет notify() или notifyAll метод для этого объекта

}

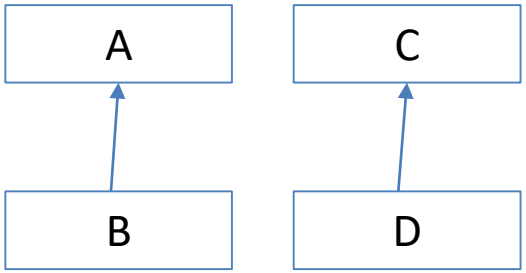
Важные моменты наследования в Java

- Java не поддерживает множественное наследование, поэтому подкласс может наследовать только один класс.
- **private**-члены суперкласса недоступные для подклассов.
- Подкласс с уровнем доступа **default** (по умолчанию) доступен другим подклассам только если они находятся в том же пакете.
- Конструкторы суперкласса не наследуются подклассами.
- Если суперкласс не имеет конструктора по умолчанию, то подкласс должен иметь явный конструктор. В противном случае он будет бросать исключение времени компиляции.
- Если в иерархии классов конструктор суперкласса требует передачи ему параметров, все подклассы должны передавать эти параметры по эстафете.
- Вызов метода **super()** всегда должен быть первым оператором, выполняемым внутри конструктора подкласса. Метод **super()** всегда ссылается на конструктор ближайшего суперкласса в иерархии.

Доступ к компонентам базового класса

super — используется как ссылка на экземпляр суперкласса с целью обеспечения доступа к одноименным нестатическим полям и методам суперкласса.

```
class A {
 float x;
 ...
}
class B extends A {
 int x;
 ...
 public void method l(int x) {
 int iX1 = x;    // присваивание значения параметра метода
 int iX2 = this.x;  // присваивание значения поля данного класса
 float fX = super.x; // присваивание значения поля суперкласса
 }
}
class C {
 ...
 public void method2() {...} ...
}
class D extends C {
 ...
 public void method2() {
  super.method2(); // вызов метода суперкласса C
 }
}
```



Конструкторы суперкласса

В производном классе конструктор суперкласса, отличный от конструктора по умолчанию, должен быть вызван явно с помощью метода **super()**.

```
public class Person {
private String name;
public Person(String name) {
this.name = name;
} }
public class Employee extends Person {
public Employee(String name) {
super(name);
} }
```

Перегрузка методов

- методы суперкласса и подкласса носят одинаковое имя но разные сигнатуры типов параметров;
- порядок следования аргументов имеет значение;
- нельзя перегрузить метод, изменяя возвращаемое значение в сигнатуре метода, этот код не скомпилируется;
- выбирается метод, для которого нужно наиболее близкое расширяющее преобразование (byte $\rightarrow$int), сужающее преобразование примитивов автоматически НЕ выполняется;
- выбор перегруженного метода осуществляется на этапе компиляции, а не на этапе выполнения

Переопределение методов

- Метод базового класса должен быть виден в производном классе (public, protected).
- Метод производного класса должен иметь тоже имя и тот же набор параметров, что и метод базового класса.
- Тип, возвращаемый методом производного класса, должен совпадать или быть подклассом типа, возвращаемого методом базового класса.
- Модификатор доступа у переопределенного метода не может быть уже метода базового класса.
- Если нужно получить доступ к версии переопределённого метода, определённого в суперклассе, то используется ключевое слово **super**.
- Выбор переопределяемого метода осуществляется на этапе выполнения.

Пример. Суперкласс с внутренним классом

```
public class Rect {
class MyRect  {  //  Внутренний класс Бесцветный прямоугольник
protected int x1,y1,x2,y2;
  public String str = "Бесцветный";
MyRect(int x1, int y1, int x2, int y2) { // конструктор
this.x1 = x1;   this.y1 = y1;   this.x2 = x2;   this.y2 = y2;
}
  public void setMyRect(int vx1, int vy1, int vx2, int vy2) {
x1 = vx1;   y1 = vy1;   x2 = vx2;   y2 = vy2;
}
  public String toString() {// вывод координат бесцветного прямоугольника
  String sz = "Прямоугольник: (" +  x1 + ", " + y1 + ", " + x2 + ", " + y2 + ")";
        return sz;
}
}
```

Продолжение примера. Описание подкласса MyColorRect

```
// Внутренний класс описывает цветные прямоугольники
class MyColorRect extends MyRect {
  protected Color rectColor=Color.white;
  public String str = "Цветной";
   MyColorRect(int x1, int y1,int x2, int y2, Color colr) { // конструктор
  super(x1, y1, x2, y2);
  rectColor = colr;
 }
  MyColorRect()     // конструктор
  {   super(0, 0, 0, 0);   }
  public void setColor(Color colr) {
   rectColor = colr;
  }
   public String toString() {// вывод координат цветного прямоугольника
   String sz = super.toString()+ rectColor.toString();
     return sz;
  }
}
```

Продолжение примера. Создание объектов базовых и производных классов

```
public static void main(String[] args) {
    MyColorRect rect1 = new Rect().new MyColorRect(0, 0, 10, 20, Color.black);
        String szStr = rect1.toString(); // из MyColorRect
        System.out.println(szStr); // черный
        MyColorRect rect2 =new Rect(). new MyColorRect();
        String szStr2 = rect2.toString(); // из MyColorRect
        System.out.println(szStr2); // белый
        MyRect rect;
        MyRect rect3=new Rect().new MyRect(1,1,2,2);
        rect=rect3;
        String szStr4 = rect.toString(); // полиморфизм из MyRect
        System.out.println(szStr4+rect.str+ rect.x1); // бесцветный
        rect=rect1; // приведение к базовому классу
        String szStr3 = rect.toString(); // полиморфизм из MyColorRect
        System.out.println(szStr3+ rect.str + rect.x1); //
        }
    }
```

Прямоугольник: (0, 0, 10, 20)java.awt.Color[r=0,g=0,b=0] сочетании красной, зеленой и синей составляющих

Прямоугольник: (0, 0, 0, 0)java.awt.Color[r=255,g=255,b=255]

Прямоугольник: (1, 1, 2, 2)Бесцветный1

Прямоугольник: (0, 0, 10, 20)java.awt.Color[r=0,g=0,b=0]Бесцветный0

Статические методы при наследовании. Для статических методов в Java полиморфизм неприменим

```
public class Book {
    public static void printReport() {
        System.out.println("Метод show() из класса Book");
    }
}
public class ProgrammerBook extends Book{
    public static void printReport() {
        System.out.println("Метод show() из класса
ProgrammerBook");
    }
}

public class BookInspector {
    public static void main(String[] args) {
        Book[] mybook = new Book[2];
        mybook[0] =  new Book();
        mybook[1] =  new ProgrammerBook();
        mybook[0]. printReport() (); // Метод show() из класса Book
        mybook[1]. printReport() (); // Метод show() из класса Book
    }
}
```

Абстрактные классы

Класс помеченный как абстрактный не может создавать объекты. Если метод не имеет тело, то его нужно пометить как abstract (класс, который содержит данный метод, должен быть тоже объявлен как абстрактный).
В производном классе этот метод должен быть переопределен.

```
// абстрактный класс фигуры
abstract class Figure{
   float x; // x-координата точки
   float y; // y-координата точки
     Figure(float x, float y){
       this.x=x;
       this.y=y;
   }
   // абстрактный метод для
получения периметра
   public abstract float
getPerimeter();
   // абстрактный метод для
получения площади
   public abstract float getArea();
}
```

```
// производный класс прямоугольника
class Rectangle extends Figure
{   private float width;
    private float height;
    Rectangle(float x, float y, float width, float height){
       // обращение к конструктору класса Figure
       super(x,y);
       this.width = width;
       this.height = height;
    }
        public float getPerimeter() {
          return width * 2 + height * 2;
    }
        public float getArea() {
            return width * height;
    }
}
```

Общий алгоритм создания простого объекта

1. Ищется класс объекта среди уже используемых в программе классов. Если его нет, то он ищется во всех доступных программе каталогах и библиотеках.
2. Выделяется память для статических полей, затем они инициализируются и вызываются блоки статической инициализации создаваемого класса (если они есть) в порядке их объявления (только один раз при первой загрузке класса).
3. Выделяется память под объект.
4. Происходит инициализация нестатических полей и блоков класса в порядке их определения. Все поля данных инициализируются своими значениями или по умолчанию (0, false или null).
5. Присваиваются аргументы конструктора переменным-параметрам для вызова этого конструктора.
6. Вызывается конструктор класса. Если конструктор начинается с явного вызова другого конструктора этого же класса (с использованием this), то нужно вычислить аргументы и выполнить этот конструктор рекурсивно.
7. Вызывается остальной код конструктора.

```
public  class Parent  {
   public static int MAX;
   public static int MIN = 100;
   public final int i = 10;
static
   {       MAX = 200;
      System.out.println("Parent::MAX = " + MAX);
   }
   public Parent()   {
      System.out.println("Parent::Parent()");
   }
  }
```

Общий алгоритм создания производного объекта

1. Ищутся классы объекта среди уже используемых в программе классов. Если их нет, то он ищется во всех доступных программе каталогах и библиотеках.
2. Выделяется память для статических полей, затем они инициализируются и вызываются блоки статической инициализации создаваемого класса (если они есть) сначала для базового класса, потом для производного класса (только один раз).
3. Выделяется память под объект, для полей базового и производного класса.
4. Происходит инициализация нестатических полей и блоков **базового класса**.
5. Присваиваются аргументы конструктора производного класса переменным-параметрам для вызова этого конструктора.
6. Вызывается конструктор базового класса.
7. Происходит инициализация нестатических полей и блоков **производного класса**.
8. Вызывается остальной код конструктора производного класса.

Принципы проектирования классов в ООП (SOLID)

1) Single responsibility principle(S) - говорит о том, что на каждый класс должна быть возложена только одна определенная обязанность. Все ресурсы, необходимые для его осуществления, должны быть инкапсулированы в этот класс и подчинены только этой задаче.

2) Open/closed principle(O) - говорит о том, что программные сущности (классы,код) должны быть открыты для расширения, но закрыты для изменений.

3) Liskov substitution principle(L) - принцип подстановки Барбары Лисков, который говорит, что функция, использующая базовый тип, должна иметь возможность использовать подтипы базового типа, не зная об этом.

4) Interface segregation princilpe(I) - говорит о том, что лучше иметь множество специализированых интерфейсов, чем один универсальный.

5) Dependency inversion principle(D) - говорит о том, что зависимости в системе должны строиться на основе абстракций. Программное обеспечение нужно разрабатывать так, чтобы различные модули были автономными и соединялись друг с другом с помощью абстракции.

Классы должны зависеть от интерфейсов или абстрактных классов, а не от конкретных классов и функций.

Рекомендации по проектированию классов Java

- Всегда храните данные в переменных, объявленных как private.
- Всегда инициализируйте данные.
- Не используйте в классе слишком много простых типов. Несколько связанных между собой полей простых типов следует объединять в новый класс.
- Не для всех полей нужно задавать методы записи (для полей, которые после создания объекта не изменяются).
- Разбивайте на части слишком крупные классы.
- Классы и методы должны иметь осмысленные имена.
- Используйте стандартную форму определения класса:
  - общедоступные элементы;
  - элементы, область видимости которых ограничена пакетом;
  - приватные элементы.