# 2. ООП

# Объектно-ориентированное программирование

- Императивное программирование — последовательность инструкций, изменяющих состояние
  - Структурное
  - Процедурное
  - Модульное
  - Объектно-ориентированное
- Декларативное программирование — описание, что надо сделать. Как это сделать — не указывается
  - Логическое
  - Функциональное

# Концепции ООП

- Абстракция — выделение существенных характеристик
- Инкапсуляция — скрытие деталей реализации
- Наследование — создание классов-потомков на основе классов-предков
- Полиморфизм — единообразная обработка различных данных

# Абстракция, объекты, классы

- Выделение существенных характеристик
- Игнорирование несущественных деталей
- Объект — программная модель реальной или абстрактной сущности
  - уникально идентифицируется (адресом или набором атрибутов)
  - атрибуты (переменные) определяют состояние
  - операции (методы) определяют поведение
  - объект — экземпляр класса
- Класс — абстрактный составной тип данных
  - данные вместе с кодом для их обработки
  - развитие понятий структуры (C) и модуля

ИТМО ВТ

# Инкапсуляция

- Класс — черный ящик
- Защита от некорректного изменения данных
  - точка на плоскости — double x, double y — всегда корректны
  - дата — int day, int month, int year — 32 мая 2015
- Открыт только интерфейс
  - Date.next(), Date.add(int days), Date.between(Date otherDate)
- Возможность изменять код, не влияя на другие объекты

# Отношения между классами

- Зависимость — "uses"
  - один класс пользуется методами другого класса
- Агрегирование — "has"
  - один класс включает в себя объекты другого класса
- Наследование — "is"
  - один класс является расширением другого класса

# Наследование

- Порождение одних классов от других
  - Потомок сохраняет переменные и методы предка
  - Потомок добавляет новые переменные и методы
  - Потомок может изменить реализацию методов предка
  - Phone { набрать номер, ответить на звонок }
  - MobilePhone = Phone + { отправить SMS, прочитать SMS }
- Построение иерархии классов
  - Реализуется отношение "is-a" (является)
  - Потомок является более специализированным предком
    - Животные — рыбы, земноводные, пресмыкающиеся, птицы, млекопитающие
    - Транспорт — поезд, самолет, корабль, автомобиль

# Полиморфизм

- Специализированный
  - System.out.println(3)
  - System.out.println("Hello!");
- Параметрический
  - Stack<T>  - push(T), T pop(), clear
- Подтипов
  - Transport.move(from, to)
    - Train.move(from,to) { реализация движения по рельсам }
    - Plane.move(from.to) { реализация движения по воздуху }
  - Transport t1 = new Train();
  - Transport t2 = new Plane();
  - t1.move(Moscow, SPB);
  - t2.move(Moscow, SPB);

# Принципы ОО проектирования

- SOLID (Robert Martin — Principles of OOD)
  - Single Responsibility Princile (единственной обязанности)
    - есть только одна причина для изменения класса
  - Open-Closed Principle (открытости-закрытости)
    - класс открыт для расширения, закрыт для изменения
  - Liskov Substitution Principle (подстановки Барбары Лисков)
    - класс должен заменяться подклассом без нарушения корректности
  - Interface Segregation Principle (разделения интерфейса)
    - класс не должен зависеть от лишних методов
  - Dependency Inversion Principle (инверсии зависимости)
    - модули верхнего уровня не должны зависеть от нижнего
    - абстракция не должна зависеть от деталей реализации
- GRASP - General Responsibility Assignment Software Patterns (Craig Larman — Applying UML and Software Patterns)
  - High Cohesion — высокая связность (внутри класса или модуля)
  - Low Coupling — низкая связанность (между классами или модулями)
  - ... и еще 7 (частично пересекаются с SOLID)

# Классы

```
class A { // объявление класса
  int field; // переменная (поле)
  A() { // конструктор
    field = 0;
  }
  void method(int par) { // метод с параметром
    int a = 1; // локальная переменная
    field = par + a;
  }
}
```

# Объекты

- объявление
  - A obj;  // obj == null
- создание — вызов конструктора (выделение памяти, присвоение значений)
  - obj = new A();
  - A obj = new A();
- ссылка на поле объекта
  - obj.field;
- вызов метода с аргументом
  - obj.method(1);
- Проверка типа
  - if (obj instanceof  A)
- удаление (сборщиком мусора)
  - obj = null;

# Модификаторы доступа

- private (в пределах класса)
- по умолчанию (в пределах пакета)
- protected (доступен в пакете и наследникам)
- public (всем)

- Реализация инкапсуляции
  - Поля — private
  - Методы — public

- Локальные переменные и параметры — без модификатора. Они доступны только внутри блока, где они определены

# Переменные (поля)

- *модификаторы тип имя;*
- private int age;
- static — переменная класса,
  - существует в единственном экземпляре для класса
  - инициализируется сразу после загрузки класса
  - может вызываться без создания экземпляров
- final — константа
  - после инициализации значение изменять нельзя
  - static final — константа класса, компилятор заменяет ее значением

# Значения полей по умолчанию

- int, byte, short = 0
- long = 0L
- char = '\u0000'
- float = 0.0F
- double = 0.0
- boolean = false
- ссылочные типы = null

- Локальные переменные не имеют значения по умолчанию

# Методы

- *модификаторы тип имя* (*тип параметр,...*) { *тело* }
- тип возвращаемого значения (если нет — void)
  - return
- a.method(b)
  - b — явный параметр
  - a — неявный параметр (ссылка на объект - this)
- static — метод класса (вызывается без ссылки на объект)
- varargs — метод с переменным числом параметров
  - method(int ... p) = method(int[] p)
  - ... - не больше одного и последний в списке

# Передача параметров

- Параметры передаются только по значению
- Внутри метода нельзя изменить значение параметра

```
public void m (int x) {
   x = x + 1;
}

y = 0;
m(y);  // после вызова метода y == 0
```

- Ссылочные типы:
  - нельзя изменить саму ссылку
  - можно изменить содержимое объекта

# Ссылка this

- this — ссылка на объект, у которого вызван метод
- this.a — ссылка на поле, если локальная переменная или параметр имеют такое же имя
- method(this) — передача ссылки на текущий объект в качестве параметра методу

# Перегрузка методов

- Методы с одинаковыми именами, но с разной сигнатурой
- Сигнатура метода — имя, количество и тип аргументов
- Тип возвращаемого значения не учитывается

println (int x)

println (long x)

println (double x)

println (char x)

println (String s)

# Метод main

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

- Точка входа в программу
- java A 100 200 300
  - A.main({"100", "200", "300"})
- Может отсутствовать
- Может использоваться для тестирования

# Конструктор

```
public class A {
    int x;
    public A() {
      x = 0;
    }
    public A(int value) {
      x = value;
    }
}

A obj = new A(4);
```

# Блоки инициализации

- статический блок — исполняется после загрузки класса

  static { }

- нестатический блок — исполняется в начале вызова любого конструктора

  { }

# Наследование

```
class A {
    int x, y;
    A() {
        x = 0; y = 0;
    }
    void draw() {
    }
}
class B extends A {
    int c;
    B() {
        c = 1;
    }
    void draw() { // переопределение метода
    }
}
```

# Ссылка super

- super — ссылка на суперкласс
- super.a — ссылка на поле суперкласса
- super.method() - вызов метода суперкласса
- super() - вызов конструктора суперкласса

# Цепочки конструкторов и неявный конструктор

- Конструкторы не наследуются
- В конструкторе можно вызвать конструктор данного класса или суперкласса с параметрами или без:

```
this();
```

```
super(x);
```

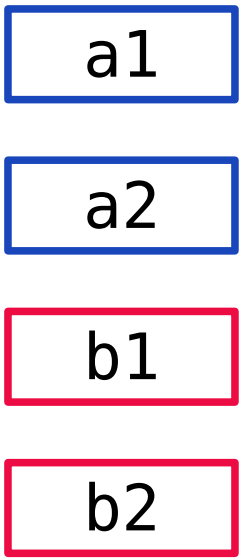
- Если такого вызова нет в первой строке, то компилятор добавит неявный вызов конструктора суперкласса **без параметров**
- Если класс не имеет конструкторов, компилятор добавляет пустой конструктор без параметров, который вызывает конструктор суперкласса.

```
public A() {
    super();
}
```

- Если в классе есть только private конструкторы — нельзя создать объект

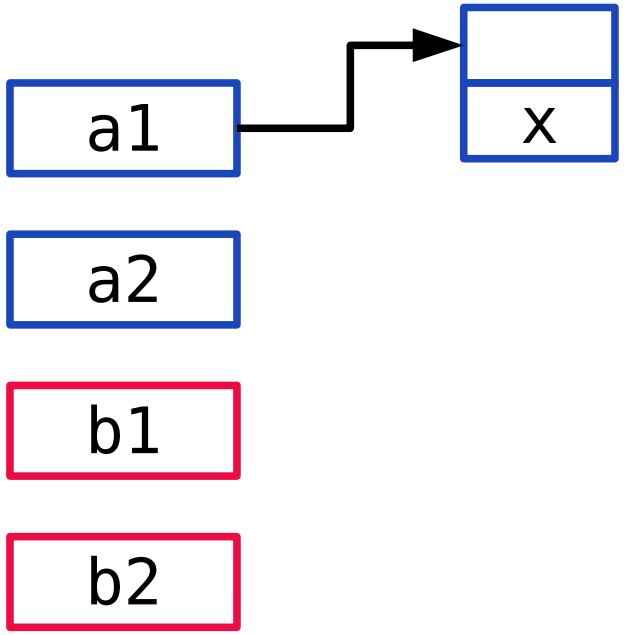
# Объекты и ссылки

```
class A { int x; }
class B extends A { double y; }

A a1, a2;
B b1, b2;
```



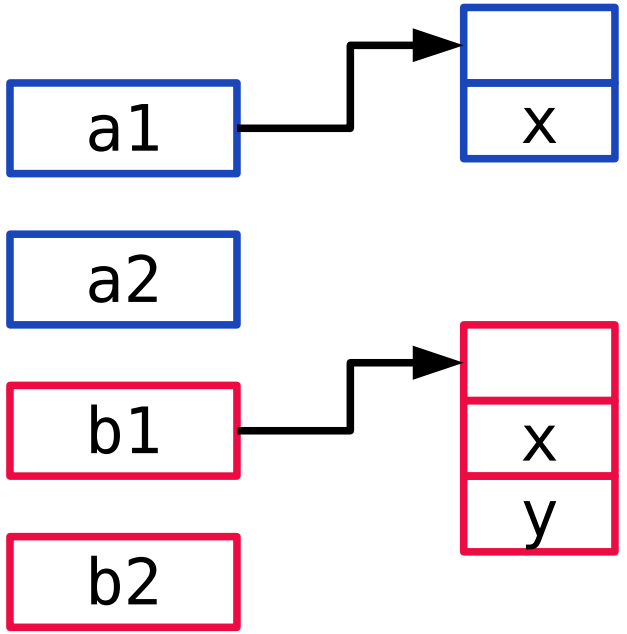
# Объекты и ссылки

```
class A { int x; }
class B extends A { double y; }

A a1, a2;
B b1, b2;
a1 = new A();
```



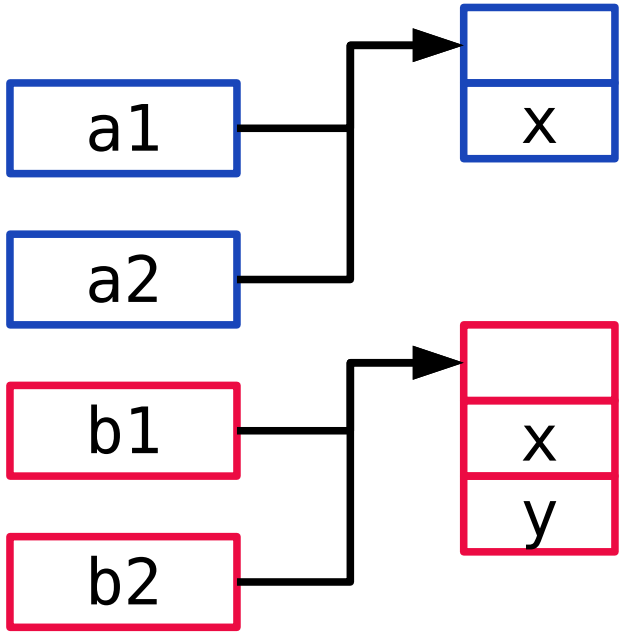
# Объекты и ссылки

```
class A { int x; }
class B extends A { double y; }

A a1, a2;
B b1, b2;
a1 = new A();
b1 = new B();
```



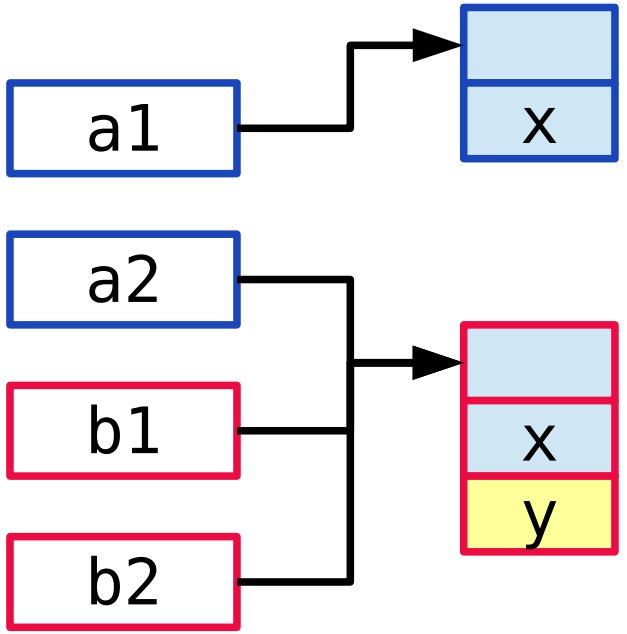
# Объекты и ссылки

```
class A { int x; }
class B extends A { double y; }

A a1, a2;
B b1, b2;
a1 = new A();
b1 = new B();
a2 = a1;
b2 = b1;
```



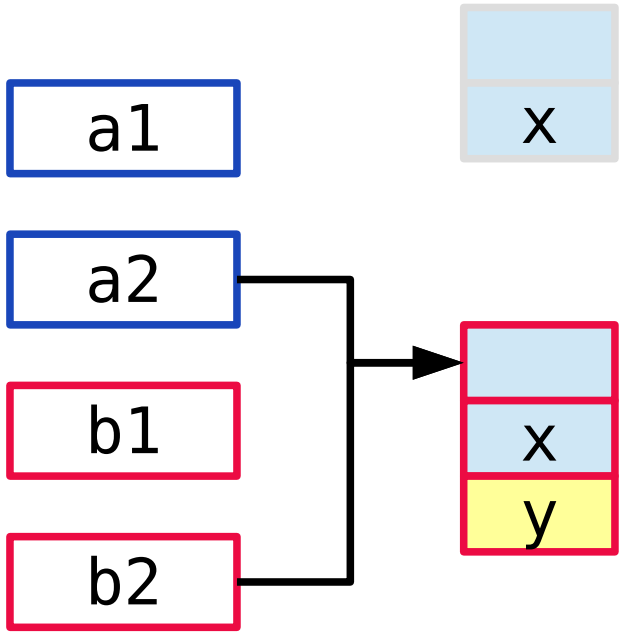
# Объекты и ссылки

```
class A { int x; }
class B extends A { double y; }

A a1, a2;
B b1, b2;
a1 = new A();
b1 = new B();
a2 = a1;
b2 = b1;
a2 = b2; // a2 = (A) b2 – неявное приведение
b2 = (B) a2; // ошибка
```



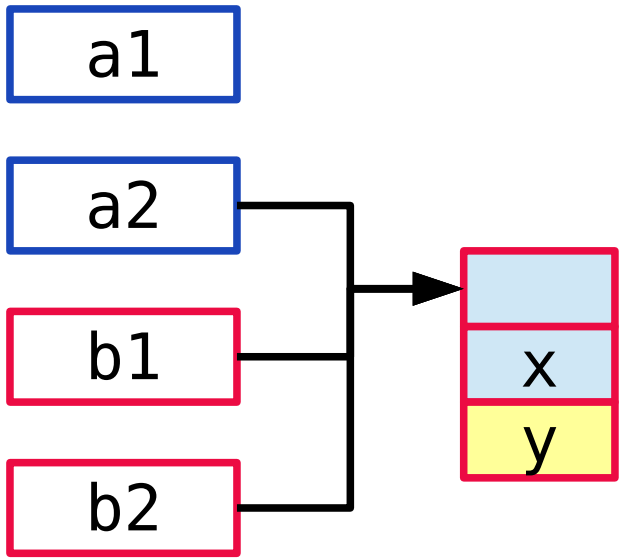
# Объекты и ссылки

```
class A { int x; }
class B extends A { double y; }

A a1, a2;
B b1, b2;
a1 = new A();
b1 = new B();
a2 = a1;
b2 = b1;
a2 = b2;
b2 = (B) a2; // ошибка
a1 = null;  // на a1 нет больше ссылок
b1 = null;
```



# Объекты и ссылки

```
class A { int x; }
class B extends A { double y; }

A a1, a2;
B b1, b2;
a1 = new A();
b1 = new B();
a2 = a1;
b2 = b1;
a2 = b2;
b2 = (B) a2; // ошибка
a1 = null;
b1 = null;
b1 = (B) a2;
```



# Наследование переменных

- Переменные private не видны классу-потомку
- Переменные protected — доступны в классе-потомке
- Переменные с тем же именем — скрыты в потомке. Доступны с помощью super

# Наследование статических методов

```
public class A {
  public static void m() { }
}
public class B extends A {
  public static void m() { }
}
```

- Статический метод с тем же именем — скрыт в потомке. Доступен с помощью super. При вызове статического метода вызывается метод предка или потомка в зависимости от типа ссылки.

  B b = new B();

  A a = b;

  b.m() // вызывается метод потомка

  a.m() // вызывается метод предка (ссылка типа A)

# Наследование нестатических методов

```
public class A {
  public void m() { }
}
public class B extends A {
  public void m() { }
}
```

- Нестатический метод с тем же именем переопределяется в потомке. При вызове происходит динамическое связывание и вызывается метод, зависящий от типа объекта, а не от типа ссылки.

  B b = new B();

  A a = b;

  b.m() // вызывается метод потомка

  a.m() // вызывается метод потомка (объект типа B)

# Динамическое связывание и полиморфизм

- Класс Figure с методом area() - подсчет площади
- Потомки класса Figure — Rectangle, Triangle, Circle с переопределенными методами area()
- Массив Figure[] figures, содержащий объекты типов Rectangle, Triangle, Circle.
- При вызове метода area() будут вызываться методы соответствующих классов:

```
for (Figure f : figures) { f.area(); }
```

- Нестатические методы — всегда виртуальные
- Выбор метода происходит динамически во время выполнения программы

# Наследование и final

- final метод — нельзя переопределять
- final класс — нельзя наследовать

# Абстрактные классы

```
abstract class A {
   int x, y;
   A() {
      x = 0; y = 0;
   }
   abstract void draw();
}
class B extends A {
   int c;
   B() {
      c = 1;
   }
   void draw() { // реализация метода
   }
}
```

# Интерфейсы

- В Java нет множественного наследования
- Интерфейс — ссылочный тип данных
- Поля только static final
- Методы без реализации — public abstract
- Методы с реализацией по умолчанию — default
- Нельзя напрямую создать объект
- Нет конструкторов
- Интерфейс может наследовать много интерфейсов
- Класс может реализовывать много интерфейсов
- Если в нескольких интерфейсах одинаковый default метод — класс должен переопределить его

# Интерфейсы

```
interface A { // public abstract
   void draw(); // public abstract
   default clear() {
      ... // реализация метода по умолчанию
   }
}
class B implements A {
   public void draw() { // реализация метода
   }
}

A obj = new B();
obj.clear();
obj.draw();
```

# Вложенные классы

```
interface X {
   void do();
}
public class A {
   private class B { // внутренний класс
   }
   A.B ab = A.new B();
   public void set() {
      class C { // локальный класс
      }
   }
   public void start(new X() { // анонимный класс
      public void do() {
      }
   }
}
```

- Вложенные классы имеют доступ ко всем элементам окружающего класса (включая private)

# Перечисляемые типы

```
public enum Days {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
  SATURDAY, SUNDAY;
}
public enum Months {
  JAN(31),FEB(28),MAR(31),APR(30),MAY(31),JUN(30),
  JUL(31),AUG(31),SEP(30),OCT(31),NOV(30),DEC(31);
private int days;
public int getDays() { return days; }
private Months(int d) { days = d; }
}
for (Days d : Days.values()) { }
```

# Аннотации

- Специальный тип интерфейса
- Стандартные аннотации:
  - @Override — переопределенный метод
  - @Deprecated — устаревший элемент
  - @SuppressWarnings — не выводить предупреждения
  - @SafeVarargs — безопасный аргумент с переменной длиной
  - @FunctionalInterface — функциональный интерфейс
- Можно создавать свои аннотации - @interface

# Пакеты

```
package ru.ifmo.cde.bars;
import java.util.*;
class A {
    java.io.Reader r;
    ArrayList v;
}
```

ru.ifmo.cde.bars.A — полное (квалифицированное) имя

./ru/ifmo/cde/bars/A.class — файл класса

javac -d . A.java

# import

- Позволяет ссылаться на классы и интерфейсы по их короткому имени

  java.util.Arrays.sort();

  import java.util.Arrays;

  Arrays.sort();

- статический import — для статических методов

  static import java.lang.Math.*;

  double a = sin(0);

  import java.lang.* - по умолчанию