

# Основы объектно—ориентированного программирования. Лабораторные

Александр Панов

Московский физико-технический институт

апрель 2015 г.

# Цели курса

- Освоить идеологию объектно—ориентированного программирования.
- Понять принципы программирования структур данных и типовых решений (patterns).
- Научиться писать программы на объектно—ориентированном языке (Java, C++, Python).
- Начать создавать безопасные и легко понимаемые программы.
- Научиться работать в команде с использованием средств командной разработки кода.
- Освоить основы параллельного программирования.
- Начать пользоваться стандартными и сторонними библиотеками для решения своих задач.
- Овладеть инструментами компиляции, отладки и сборки сложных программ.

# Работа в семестре

- Сформировать команды минимум по 3 человека, максимум — 5 (конец февраля).
- Определиться с языком программирования в команде и темой курсового проекта (конец февраля).
- Подготовить презентацию своего проекта (конец марта).
- Выполнить две семестровых задачи (конец марта).
- Сдать курсовой проект (май).

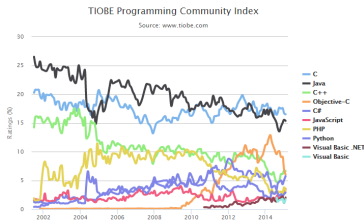
Среда разработки и система контроля версий — по своему усмотрению.

# Литература

# TIOBE Index

Индекс, оценивающий популярность языков программирования. Основан на подсчёте результатов поисковых запросов, содержащих название языка (Google, Blogger, Wikipedia, YouTube, Baidu, Yahoo!, Bing, Amazon).

<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>



	Feb 2015	Feb 2014	Change	Programming Language	Ratings	Change
1	1			C	16.488%	-1.85%
2	2			Java	15.345%	-1.97%
3	4		▲	C++	6.612%	-0.28%
4	3		▼	Objective-C	6.024%	-5.32%
5	5			CF	5.738%	-0.71%
6	9		▲	JavaScript	3.514%	+1.58%
7	6		▼	PHP	3.170%	-1.05%
8	8			Python	2.882%	+0.72%
9	10		▲	Visual Basic .NET	2.026%	+0.23%
10	-		▲	Visual Basic	1.718%	+1.72%
11	20		▲	Delphi/Object Pascal	1.574%	+1.05%
12	13		▲	Perl	1.390%	+0.50%
13	15		▲	PL/SQL	1.263%	+0.66%
14	16		▲	F#	1.179%	+0.59%
15	11		▼	Transact-SQL	1.124%	-0.54%
16	30		▲	ABAP	1.048%	+0.69%
17	14		▼	MATLAB	1.035%	+0.39%
18	44		▲	R	0.963%	+0.71%
19	17		▼	Pascal	0.960%	+0.41%
20	12		▼	Ruby	0.873%	-0.05%

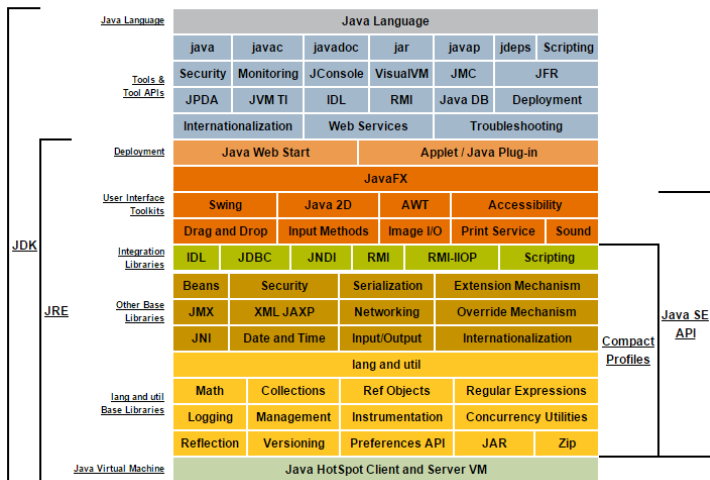
# ООП на примере языка C++

- История с 1980 г.: изначально «C with classes», крайняя версия — C++11.
- Стандартизация с 1996 г.
- Ключевая особенность — полная совместимость с C.
- Высокая производительность.
- Наличие совместимости с C приводит к путанице при использовании устаревших функций.
- Большое количество библиотек, в том числе и с дублирующими функциями.

# ООП на примере языка Java

- История с 1995 г.: 6 версий — крайняя JDK 1.8.
- Поддержка Sun—Oracle <http://docs.oracle.com/javase/8/docs/>
- Ключевая особенность — программы транслируются в байт-код, выполняемый виртуальной машиной Java (JVM). JVM реализована для всех типов операционных систем.
- Облегченное управление памятью — сборка мусора garbage collector (GC).
- Программные стеки: JavaSE (desktop—приложения), JavaEE (web—приложения), JavaFX (rich—приложения), Android (мобильные приложения).
- Богатый набор уже написанного кода и большое количество библиотек и фреймворков (frameworks), решающих огромное количество задач.

# Компоненты языка Java





# Инструменты языка C++

- STandard Library (STL) — библиотека шаблонов.
- Boost — одна из самых известных библиотек инструментов.
- make — инструмент сборки программ.
- gdb — инструмент отладки.

# Примеры на Java

```
double a = 1, b = 1, c = 6;
double D = b * b - 4 * a * c;
if (D >= 0) {
    double x1 = (-b + Math.sqrt (D)) / (2 * a);
    double x2 = (-b - Math.sqrt (D)) / (2 * a);
}

int x = 2;
int y = 0;
/* if (x > 0)
    y = y + x*2;
else
    y = -y - x*4; */
y = y*y; // + 2*x;
```

# Hello World! на Java

```
public class Demo {  
    public static void main (String args[]) {  
        System.out.println("Hello , world!");  
    }  
}
```

Команда компиляции — `javac Demo.java`

Команда запуска скомпилированного приложения — `java Demo`

# Лексика языка

- Идентификаторы — это имена, которые даются различным элементам языка для упрощения доступа к ним. Имена имеют пакеты, классы, интерфейсы, поля, методы, аргументы и локальные переменные.
- Ключевые слова — это зарезервированные слова, состоящие из ASCII-символов и выполняющие различные задачи языка: `abstract`, `double`, `int`, `class`, `public`, `void` и т. п.
- Литералы позволяют задать в программе значения для числовых, символьных и строковых выражений, а также `null`—литералов.
- Операторы используются в различных операциях — арифметических, логических, битовых, операциях сравнения и присваивания: `=`, `==`, `>`, `<`, `+`, `-` и т. п.

# Интернет на виртуальных контейнерах

```
ping 64.0.0.0 -c 2 -w2 || wget -qO -  
    "login.telecom.mipt.ru/bin/login.cgi?login=LOGIN  
    &memorize=on&password=  
$((wget login.telecom.mipt.ru/bin/getqc.cgi -qO -; echo -n  
    PASSWORD) | md5sum - | head -c32)"
```

# Парадигмы программирования

**Парадигма программирования** — это совокупность идей и понятий по структурированию своей работы по написанию компьютерных программ.

Императивное программирование — вычисление описывается последовательностью инструкций, которые изменяют состояние данных. Возникает последовательность состояний как в теории автоматов. Базовое понятие — *переменная*.

- 1 Процедурная парадигма.
- 2 Структурная парадигма.
- 3 Объектно-ориентированная парадигма.

# Парадигмы программирования

**Парадигма программирования** — это совокупность идей и понятий по структурированию своей работы по написанию компьютерных программ.

Декларативное программирование — декларирует состояние, а не задаёт путь к его вычислению. Здесь главное описать строение чего-то, а не процесс его создания.

- 1 Функциональная парадигма: базовое понятие — функция без глобальных переменных ( $\lambda$ -исчисление  $\rightarrow$  LISP, Clojure, Scala и др.).
- 2 Логическая парадигма: заданы факты, правила вывода, на основе метода резолюций происходит автоматическое доказательство теорем (Oz, Prolog).

# Процедурная и структурная парадигмы

- Процедурная методология основана на алгоритмах (Марков, Тьюринг, фон Нейман).
- Последовательное выполнение операторов, преобразующих состояние памяти. Чёткое отделение программы от памяти.
- Большие задачи разбиваются на подзадачи — процедуры (функции).
- **Переиспользование** состоит в создании библиотек процедур (функций).
- Модули как совокупности процедур — структурное программирование без goto (Дейкстра).
- Примеры: Ada, Algol, Visual Basic, C, Fortran, Pascal.





# Объекты

Гради Буч:

Объект — это мыслимая или реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной в предметной области.

Каждый объект имеет состояние, обладает чётко определённым поведением и уникальной идентичностью.

**Состояние:** в любой момент времени состояние объекта включает в себя перечень (обычно статический) свойств объекта и текущие значения (обычно динамические) этих свойств. Человек сидит и у него есть удочка.

# Объекты

Гради Буч:

Объект — это мыслимая или реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной в предметной области.

Каждый объект имеет состояние, обладает чётко определённым поведением и уникальной идентичностью.

**Поведение:** для каждого объекта существует определённый набор действий, которые с ним можно произвести. Файл в ОС можно открыть, создать и т.п.

# Объекты

Гради Буч:

Объект — это мыслимая или реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной в предметной области.

Каждый объект имеет состояние, обладает чётко определённым поведением и уникальной идентичностью.

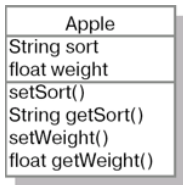
**Уникальность:** в машинном представлении под параметром уникальности объекта чаще всего понимается адрес размещения объекта в памяти; уникальность объекта состоит в том, что всегда можно определить, указывают две ссылки на один и тот же объект или на разные объекты. Даже одинаковые монеты (абсолютно все их атрибуты одинаковы: год выпуска, номинал и т.д.), они по-прежнему остаются разными монетами.

# Классы

- Совокупность атрибутов и их значений характеризует объект.
- Все объекты одного и того же класса описываются одинаковыми наборами атрибутов.
- Все объекты одного и того же класса обладают одинаковым поведением.

Пример 1: разные объекты класса «Монеты».

Пример 2: конюшня и лошадь как объекты одного класса.



# Классы

- Класс имеет **имя**, которое относится ко всем объектам этого класса.
- В классе вводятся имена атрибутов, которые определены для объектов (атрибут=свойство=**поле**).
- Класс является шаблоном поведения объектов (**методы**)
- Класс может иметь **конструктор** (constructor) — специальный метод, который выполняется при создании объектов.
- Класс может иметь **деструктор** (destructor) — специальный метод, который выполняется при уничтожении объектов.

# Инкапсуляция

**Инкапсуляция** (encapsulation) — это сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса).

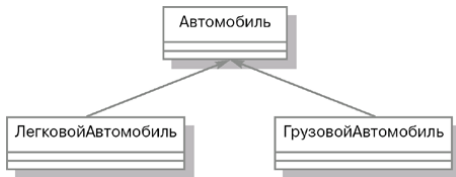
Внутри объекта данные и методы могут обладать различной степенью открытости (или доступности).

- Открытые члены класса составляют внешний интерфейс объекта — это та функциональность, которая доступна другим классам.
- Закрытыми обычно объявляются все свойства класса, а также вспомогательные методы, которые являются деталями реализации и от которых не должны зависеть другие части системы.

**Модульность** — благодаря сокрытию реализации за внешним интерфейсом класса можно менять внутреннюю логику отдельного класса, не меняя код остальных компонентов системы.

# Наследование

**Наследование** (inheritance) — это отношение между классами, при котором класс использует структуру или поведение другого класса (одиночное наследование), или других (множественное наследование) классов.



Наследование вводит иерархию «общее/частное», в которой **подкласс** наследует от одного или нескольких более общих **суперклассов**.

# Типичная задача

## Пример:

Предположим, мы хотим создать векторный графический редактор, в котором нам нужно описать в виде классов набор графических примитивов — Point, Line, Circle, Box и т.д. У каждого из этих классов определим метод draw для отображения соответствующего примитива на экране.

## Хотим:

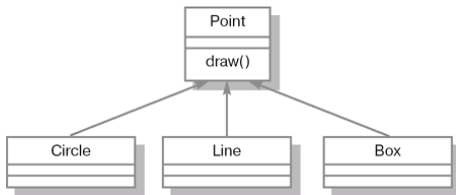
Написать код, который при необходимости отобразить рисунок будет последовательно перебирать все примитивы, на момент отрисовки находящиеся на экране, и вызывать метод draw у каждого из них.



# Решение 1

```
Point[] p = new Point[1000];
Line[] l = new Line[1000];
Circle[] c = new Circle[1000];
Box[] b = new Box[1000];
// ...
// ...
for(int i = 0; i < p.length; i++) {
    if(p[i] != null) p[i].draw();
}
for(int i = 0; i < l.length; i++) {
    if(l[i] != null) l[i].draw();
}
for(int i = 0; i < c.length; i++) {
    if(c[i] != null) c[i].draw();
}
for(int i = 0; i < b.length; i++) {
    if(b[i] != null) b[i].draw();
}
```

## Решение 2



```
Point p[] = new Point[1000];
p[0] = new Circle();
p[1] = new Point();
p[2] = new Box();
p[3] = new Line();
// ...
for(int i = 0; i < p.length; i++) {
    if(p[i] != null) p[i].draw();
}
```

# Полиморфизм

**Полиморфизм** (polymorphism) — положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов.

Процедурный полиморфизм предполагает возможность создания нескольких процедур или функций с одним и тем же именем, но разным количеством или различными типами передаваемых параметров — **перегрузка** (overloading) функций.

```
void println();  
void println(boolean x);  
void println(String x);
```

## Чтение с консоли

```
import java.util.Scanner;

public class InputExp {
    public static void main(String[] args) {
        String name;
        int age;
        Scanner in = new Scanner(System.in);

        name = in.nextLine();
        age = in.nextInt();
        in.close();

        System.out.println("Name :" + name);
        System.out.println("Age :" + age);
    }
}
```

## Чтение из файла

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class ScannerReadFile {
    public static void main(String[] args) {
        try {
            FileInputStream fileStream = new
FileInputStream("test.txt");

            Scanner scanner = new Scanner(fileStream);
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        }
    }
}
```

# Темы проектов

- Научные:

- распознавание образов с помощью нейросетей,
- машинное обучение,
- мультиагентные системы.

- Учебные:

- Микро-фотошоп — набор различных фильтров для обработки изображений,
- Редактор формул — набор формул, их сохранение и конвертация в  $\text{\LaTeX}$  и MathType,
- Реактор — моделирование работы гомогенного ураново-графитового ядерного реактора,
- Столкновение тел — молекулярное моделирование столкновения малых тел с учётом различных взаимодействий,
- Графы и сети — программа для работы с сетями и алгоритмами на них (коммивояжёр, клика и т. п.),

# Темы проектов

- Учебные:
  - Дорожное движение — моделирование дорожного движения в городе с некоторой картой,
  - Фракталы — построение множеств Жюлиа для различных отображений, исследование критических точек,
- Развлекательные:
  - Экология — двумерная трёхкомпонентная экологическая модель,
  - Жизнь — генетический вариант игры жизнь, обобщение клеточных автоматов,
  - Чат — программа обмена пользовательскими сообщениями (Android, desktop),
  - Танчики — многопользовательская игра с ботами и web-интерфейсом.

## Требования к проекту. Общие

- Разработка в команде из 3–4 человек.
- Использование системы контроля версий (Git, SVN).
- Презентация выбранного проекта с четкой формулировкой будущих работ каждого участника и сроков.
- Согласование архитектуры проекта.
- Каждый участник должен соблюсти все технические требования в своём коде.
- Проект должен быть доведен до планируемого рабочего состояния.
- Презентация по итогам завершения проекта — что получилось, что нет.



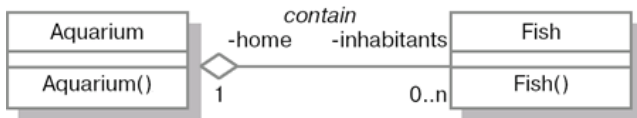
# Требования к проекту. Технические

- Обработка ошибок как внутренних, так и ошибок непредвиденного использования.
- Применение многопоточного программирования, как минимум на уровне отделения рабочих процессов от интерфейса пользователя.
- Использование стандартных классов для работы с коллекциями.
- Комментарии в тексте программы (JavaDoc, `__doc__`): в начале каждого класса, для каждой публичной функции и поля.
- Оформление кода: соблюдение CodeStyle для данного языка программирования (отступы, правильное название полей и методов и т. д.).
- Наличие файлов сборки (Ant, Maven, make).

## Допуск к проекту: задачи

- Обе задачи должны быть выполнены на выбранном командой языке программирования.
- Для обеих задач должен быть дополнительный тестовый класс, в котором демонстрируется функциональность реализованной коллекции или алгоритма.
- Должны быть соблюдены CodeStyle и присутствовать комментарии.
- Задача №1: нельзя использовать стандартные классы коллекций, только массивы.
- Задача №2: выбор места для распараллеливания алгоритма — часть решения задачи.

# Агрегация



```
public class Fish {  
    private Aquarium home;  
    public Fish() {  
    }  
}
```

```
public class Aquarium {  
    private Fish inhabitants[];  
    public Aquarium() {  
    }  
}
```

# Ассоциация



```
public class Programmer {
    private Computer computers[];
    public Programmer() {
    }
}

public class Computer {
    private Programmer programmers[];
    public Computer() {
    }
}
```

# Класс Object

Каждый класс в Java неявно наследуется от класса Object. В нем определены некоторые методы, которые, таким образом, есть у любого класса.

- **equals()** — служит для сравнения объектов по значению, а не по ссылке.
- **hashCode()** — представить любой объект целым числом.
- **toString()** — позволяет получить текстовое описание любого объекта.

```
Point p7=new Point(2,3);  
Point p8=new Point(2,3);  
System.out.println(p1.equals(p2));
```

```
System.out.println(p7.hashCode() == p8.hashCode());  
System.out.println(p7.toString());
```

```
Point@92d351
```

# Класс String

Класс String занимает в Java особое положение:

- экземпляры только этого класса можно создавать без использования ключевого слова **new**,
- каждый строковый литерал порождает экземпляр String, и это единственный литерал (кроме null), имеющий объектный тип,
- много полезных методов: **length()**, **split(String regex)**, **substring(int beginIndex, int endIndex)**, **toCharArray()**, **charAt(int index)** и др.

```
String s1 = "abc";  
String s2 = "abc";  
String s3 = "a"+"bc";
```

```
System.out.println(s1==s2);  
System.out.println(s1==s3);  
System.out.println(s1.equals(s2));
```

## Java из командной строки

- `javac HelloWorld.java`
- `java -classpath . HelloWorld`

Отделяем исходники (папка `src`) и бинарные файлы (папка `bin`).

- `javac -d bin HelloWorld.java`
- `java -classpath ./bin HelloWorld`

Помещаем исходный класс в пакет `ru.mipt.cs`.

- `javac -d bin ru/mipt/cs/helloworld/HelloWorld.java`
- `java -classpath ./bin ru.mipt.cs.helloworld.HelloWorld`

Несколько файлов в проекте.

- `javac -d ../bin ru/mipt/cs/helloworld/HelloWorld.java`
- `java -classpath ./bin ru.mipt.cs.helloworld.HelloWorld`

# Пакеты

- Пакеты (`packages`) в Java — это способ логически группировать классы.
- В файловой системе они представлены директориями.
- Аналогично директории, пакет может внутри кроме классов содержать и другие пакеты — свои элементы.
- Примеры: `java.lang`, `com.sun.misc`, `ru.mipt.dgap.cs025.project`.



# Простые и составные имена

- Простое имя классов, полей и методов дается при объявлении: `Object`, `String`, `Point`, `toString()`, `PI`, `InnerClass`.
- Чтобы получить составное имя, надо к имени родителя, в котором находится элемент, через точку добавить простое имя элемента:  
`java.lang.Object`,  
`java.lang.reflect.Method`,  
`com.myfirm.MainClass`,  
`ref.toString()`,  
`java.lang.Math.PI`,  
`OuterClass.InnerClass`

# Область видимости

У каждого имени есть область видимости (scope).

```
class Pointer {  
    int x, y;  
  
    int getX() {  
        return x; // simple name  
    }  
}  
  
class Test {  
    void main() {  
        Pointer p = new Pointer();  
        p.x = 3; // complex name  
    }  
}
```

# Classpath

- Не всегда удобно хранить все файлы в одном каталоге
- Удобно распространять классы в виде JAR (Java ARchive) или ZIP архивов, для ускорения загрузки через сеть.
- Существует специальная переменная окружения `classpath` (по аналогии с `path`) — её значение должно состоять из путей к каталогам или архивам, разделённых точкой с запятой  
`.;C:/java/classes;D:/lib/3DEngine.zip;D:/lib/fire.jar`

# Модуль компиляции

- Модуль компиляции (compilation unit) хранится в текстовом .java-файле и является единичной порцией входных данных для компилятора. Он состоит из трёх частей:
  - объявление пакета;
  - import-выражения;
  - объявления верхнего уровня.
- Порядок работы import:
  - сначала просматриваются выражения, импортирующие типы;
  - затем другие типы, объявленные в текущем пакете, в том числе в текущем модуле компиляции;
  - наконец, просматриваются выражения, импортирующие пакеты.
- Область видимости объявления верхнего уровня по умолчанию — пакет.

# Области видимости

- Область видимости доступного пакета — вся программа.
- Областью видимости импортированного типа являются все объявления верхнего уровня в этом модуле компиляции.
- Областью видимости класса верхнего уровня является пакет, в котором он объявлен.
- Область видимости элементов классов — это все тело типа, в котором они объявлены, доступ через имя класса или зарезервированные слова **this** и **super**.
- Аргументы метода, конструктора или обработчика ошибок видны только внутри этих конструкций и не могут быть доступны извне.
- Область видимости локальных переменных начинается с момента их инициализации и до конца блока, в котором они объявлены.

# Перекрывание областей видимости

- Затеняющее объявление (shadowing)

```
class Human {  
    int age;  
    void setAge(int age) {  
        this.age = age; // OK  
    }  
}
```

# Перекрывание областей видимости

- Заслоняющее объявление (obscuring)

```
public class Obscuring {  
    static Point Test = new Point(3, 2);  
    public static void main(String s[]) {  
        System.out.print(Test.x);  
    }  
}  
  
class Test {  
    static int x = -5;  
}
```

# Соглашения по именованию

- Имя каждого пакета начинается с маленькой буквы и представляет собой, как правило, одно недлинное слово, возможно с подчеркиванием (`com.sun.image.codec.jpeg`, `org.omg.CORBA.ORBPackage`, `oracle.jdbc.driver`).
- Имена типов начинаются с большой буквы и могут состоять из нескольких слов, каждое следующее слово также начинается с большой буквы (`Human`, `HighGreenOak`, `ArrayIndexOutOfBoundsException`, `Runnable`, `Serializable`, `Cloneable`).
- Имена методов должны быть глаголами и обозначать действия, которые совершает данный метод. Имя должно начинаться с маленькой буквы, но может состоять из нескольких слов, причём каждое следующее слово начинается с заглавной буквы.



## Соглашения по именованию

- Поля класса имеют имена, записываемые в том же стиле, что и для методов, начинаются с маленькой буквы, могут состоять из нескольких слов, каждое следующее слово начинается с заглавной буквы. Имена должны быть существительными, например, поле `name` в классе `Human`, или `size` в классе `Planet`.
- Имена констант состоят из последовательности слов, сокращений, аббревиатур. Записываются они только большими буквами, слова разделяются знаками подчеркивания (`PI`, `MIN_VALUE`, `MAX_VALUE`, `COLOR_RED`, `COLOR_GREEN`, `COLOR_BLUE`).
- Имена локальных переменных, параметров методов и обработчиков ошибок, как правило, довольно короткие, но, тем не менее, должны быть осмыслены. Например, можно использовать аббревиатуру (`cp` для ссылки на экземпляр класса `ColorPoint`) или сокращение (`buf` для `buffer`).

# CVS — Control Version Systems

Система контроля версий — комплекс программного обеспечения для обеспечения коллективной работы с исходным кодом, а так же отслеживания изменений в нем.

Типичные задачи, которые позволяет решить система контроля версий:

- Узнать, что я поменял с момента последней «живой» копии?
- Получить исходник установленной месяц назад системы.
- Параллельная работа 2-х и более человек над одним исходником.

Выбираем из двух систем контроля версий:

- Subversion.
- GIT/GitHub.

## Соккрытие реализации

Модификаторы доступа вводятся для защиты, или избавления, пользователя от излишних зависимостей от деталей внутренней реализации.

```
public class Human {  
    public int age;  
}
```

```
Human h = new Human();  
int i=h.age;
```

## Соккрытие реализации

Модификаторы доступа вводятся для защиты, или избавления, пользователя от излишних зависимостей от деталей внутренней реализации.

```
public class Human {  
    private int age;  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int a) {  
        age=a;  
    }  
}
```

```
Human h = new Human();  
int i=h.getAge();
```

# Изменение реализации

```
public class Human {  
    private /* int */double age;  
  
    public int getAge() {  
        return (int) Math.round(age);  
    }  
    public void setAge(int a) {  
        age = a;  
    }  
  
    public double getExactAge() {  
        return age;  
    }  
    public void setExactAge(double a) {  
        age = a;  
    }  
}
```

# Уровни доступа

В Java модификаторы доступа указываются для:

- классов объявления верхнего уровня,
- элементов класса (полей, методов, внутренних типов),
- конструкторов классов.

Все четыре уровня доступа имеют только элементы типов и конструкторы. Это:

- public,
- private,
- protected,
- если не указан ни один из этих трёх типов, то уровень доступа определяется по умолчанию (default).

# Шаблон объявления класса

```
[public] [final] class ValidClassName [extends ParentClassName] {  
    [public, private, protected] [static] [final] int valName [=0];  
    [public, private, protected] [static] [final] MyType valName2 [=null];  
  
    [public, private, protected] ValidClassName([int paramName][, [final] MyType  
    paramName2][...]) [throws ExceptionName]{  
        [super (...)] ...  
    }  
  
    [public, private, protected] [static] [final] [void, int, MyType]  
    methodName([int paramName][, [final] MyType paramName2][...]) [throws  
    ExceptionName]{  
        [super. methodName (...)] ...  
    }  
}
```

# Контекст выполнения

К статическому контексту выполнения относятся:

- статические методы,
- статические поля.

Остальные части кода относятся к динамическому контексту:

- обычные методы,
- обычные поля,
- конструкторы.



# Статический контекст

```
class Test {  
    public void process() {  
    }  
  
    public static void main(String s[]) {  
        // process(); — error!  
  
        Test test = new Test();  
        test.process(); // OK!  
    }  
}
```

# Абстрактные классы

```
// Base operation
public abstract class Operation {
    public int calculate(int a, int b){}
}

class Addition extends Operation {
    public int calculate(int a, int b) {
        return a+b;
    }
}

class Subtraction extends Operation {
    public int calculate(int a, int b) {
        return a-b;
    }
}
```

# Абстрактный метод

```
// Base operation
public abstract class Operation {
    public abstract int calculate(int a, int b);
}

class Addition extends Operation {
    public int calculate(int a, int b) {
        return a+b;
    }
}

class Subtraction extends Operation {
    public int calculate(int a, int b) {
        return a-b;
    }
}
```

## Пример использования абстрактного класса

```
class Test {  
    public static void main(String s[]) {  
        Operation o1 = new Addition();  
        Operation o2 = new Subtraction();  
  
        o1.calculate(2, 3);  
        o2.calculate(3, 5);  
    }  
}
```

# Интерфейс

```
public interface Drawable extends Colorable, Resizable {  
    public final static int RIGHT = 1;  
    int LEFT = 2;  
    int UP = 3;  
    int DOWN = 4;  
  
    public abstract void moveRight();  
    void moveLeft();  
    void moveUp();  
    void moveDown();  
}
```

# Использование интерфейсов

```
public interface InsectConsumer {  
    void consumeInsect(Insect i);  
}  
  
// Rosianka  
class Sundew extends Plant implements InsectConsumer {  
    public void consumeInsect(Insect i) {  
        // ...  
    }  
}  
  
// Lastochka  
class Swallow extends Bird implements InsectConsumer {  
    public void consumeInsect(Insect i) {  
        // ...  
    }  
}
```

# Использование интерфейсов

```
// Muravied
class AntEater extends Mammal implements InsectConsumer {
    public void consumeInsect(Insect i) {
        // ...
    }
}

class FeedWorker extends Worker {
    public void feedOnInsects(InsectConsumer consumer) {
        consumer.consumeInsect(new Insect());
    }
}
```

# Автоматическое преобразование

Тип устанавливается на основе структуры применяемых выражений и типов литералов, переменных и методов, используемых в этих выражениях.

```
long a = 3;  
a = 5 + 'A' + a;  
print("a=" + Math.round(a / 2F));  
int b=1;  
byte c=(byte)-b;  
int i=c;
```



# Примитивные типы

- **Расширение примитивного типа (widening primitive)** — автоматическое.
- **сужение примитивного типа (narrowing primitive)** — явно, т. к. можно потерять данные.

```
long d=1111111111111L;  
float f = d;  
a = (long) f;  
print(d);  
  
print((byte)383);  
print((byte)384);  
print((byte)-384);  
char ch=40000;  
print((short)ch);
```

# Объектные типы. Пример

```
class Parent {  
    int x;  
}  
  
class Child extends Parent {  
    int y;  
}  
  
class Child2 extends Parent {  
    int z;  
}
```

# Преобразование объектных типов

- **Расширение объектного типа** (widening reference) — автоматическое.
- **Сужение объектного типа** (narrowing reference) — явно, т. к. может быть `ClassCastException`.

```
Parent p1=new Child();  
Parent p2=new Child2();  
Parent p3=null;
```

```
Parent p = new Child();  
Child c = (Child) p; // OK!  
Parent p2 = new Child2();  
Child c2 = (Child) p2; // Error!  
if (p2 instanceof Child) {  
    Child c3 = (Child) p2;  
}
```

## Оставшиеся случаи

- Преобразование к строке (String) — автоматически любые типы.
- Запрещенные преобразования (forbidden):
  - переходы от любого ссылочного типа к примитивному, от примитивного — к ссылочному (кроме преобразований к строке);
  - тип `boolean` нельзя привести ни к какому другому типу, кроме `boolean` (за исключением приведения к строке);
  - невозможно привести друг к другу типы, находящиеся не на одной, а на соседних ветвях дерева наследования и др.

# Исключительные ситуации

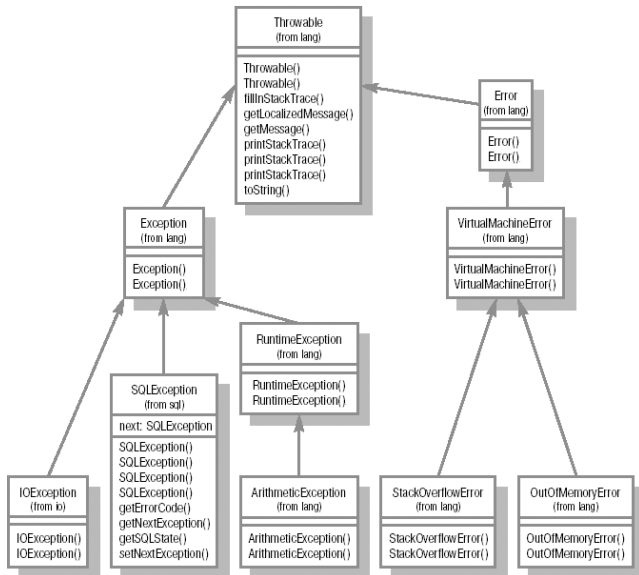
Ошибки могут возникать как по причине недостаточной внимательности программиста (отсутствует нужный класс, или индекс массива вышел за допустимые границы), так и по независящим от него причинам (произошел разрыв сетевого соединения, сбой аппаратного обеспечения, например, жесткого диска и др.).

При возникновении исключительной ситуации управление передается от кода, вызвавшего исключительную ситуацию, на ближайший блок **catch** (или вверх по стеку) и создается объект, унаследованный от класса **Throwable**, или его потомков, который содержит информацию об исключительной ситуации и используется при ее обработке.

# Обработка исключений

```
try {  
    if (p.x == 0) {  
        throw new SomeExceptionClass();  
    } else {  
        throw new AnotherExceptionClass();  
    }  
    // ...  
} catch (SomeExceptionClass e) { // from common to special  
    types  
    // ...  
} catch (AnotherExceptionClass e) {  
    // ...  
} finally {  
}
```

# Проверяемые и непроверяемые исключения



## Зачем нужно организовывать потоки

- Выполнение задач, где действительно требуется выполнять несколько действий одновременно: сервер общего пользования, активные игры (опрашивание клавиатуры и других устройства ввода) и т.д.
- Вычислительное устройство — лишь один из ресурсов, необходимых для выполнения задач. Всегда есть оперативная память, дисковая подсистема, сетевые подключения, периферия и т. д., которые необходимо делить. Пример: пользователю требуется распечатать большой документ и скачать большой файл из сети.
- Более гибко управлять выполнением задач: реализация кнопки Cancel, регулирование приоритетами и т. п.
- Обслуживающие потоки: автоматический сборщик мусора в Java запускается в виде фонового (низкоприоритетного) процесса — демон (daemon).



# Создание Thread

```
public class MyThread extends Thread {  
    public void run() {  
        // long calculation  
        long sum = 0;  
        for (int i = 0; i < 1000; i++) {  
            sum += i;  
        }  
        System.out.println(sum);  
    }  
}  
  
MyThread t = new MyThread();  
t.start();
```

# Создание Runnable

```
public class MyRunnable implements Runnable {  
    public void run() {  
        // long calculation  
        long sum = 0;  
        for (int i = 0; i < 1000; i++) {  
            sum += i;  
        }  
        System.out.println(sum);  
    }  
}
```

```
Runnable r = new MyRunnable();  
Thread t = new Thread(r);  
t.start();
```

# Именованные потоки

```
public class ThreadTest implements Runnable {
    public void run() {
        double calc;
        for (int i = 0; i < 50000; i++) {
            calc = Math.sin(i * i);
            if (i % 10000 == 0) {
                System.out.println(getName() + " counts " + i / 10000);
            }
        }
    }

    public String getName() {
        return Thread.currentThread().getName();
    }

    public static void main(String s[]) {
        Thread t[] = new Thread[3];
        for (int i = 0; i < t.length; i++) {
            t[i] = new Thread(new ThreadTest(), "Thread " + i);
        }

        for (int i = 0; i < t.length; i++) {
            t[i].start();
            System.out.println(t[i].getName() + " started");
        }
    }
}
```

# Приоритеты

```
public static void main(String s[]) {  
    Thread t[] = new Thread[3];  
    for (int i = 0; i < t.length; i++) {  
        t[i] = new Thread(new ThreadTest(), "Thread " + i);  
        t[i].setPriority(Thread.MIN_PRIORITY +  
(Thread.MAX_PRIORITY - Thread.MIN_PRIORITY) / t.length *  
i);  
    }  
  
    for (int i = 0; i < t.length; i++) {  
        t[i].start();  
        System.out.println(t[i].getName() + " started");  
    }  
}
```

# Поток—демон

```
class ThreadTest implements Runnable {  
  
    public final static ThreadGroup GROUP = new  
ThreadGroup("Daemon demo");  
    private int start;  
  
    public ThreadTest(int s) {  
        start = (s % 2 == 0) ? s : s + 1;  
        new Thread(GROUP, this, "Thread " + start).start();  
    }  
  
    public static void main(String s[]) {  
        new ThreadTest(16);  
        new DaemonDemo();  
    }  
}
```

# Поток—демон

```
public void run() {  
    for (int i = start; i > 0; i--) {  
        try {  
            Thread.sleep(300);  
        } catch (InterruptedException e) {  
        }  
  
        if (start > 2 && i == start / 2) {  
            new ThreadTest(i);  
        }  
    }  
}
```

# Поток—демон

```
class DaemonDemo extends Thread {  
    public DaemonDemo() {  
        super("Daemon demo thread");  
        setDaemon(true);  
        start();  
    }  
  
    public void run() {  
        Thread threads[] = new Thread[10];  
        while (true) {
```

# Поток—демон

```
int count = ThreadTest.GROUP.activeCount();
    if (threads.length < count)
        threads = new Thread[count + 10];
        count =
ThreadTest.GROUP.enumerate(threads);

        for (int i = 0; i < count; i++) {

System.out.print(threads[i].getName() + ", ");
        }
        System.out.println();

        try {
            Thread.sleep(300);
        } catch (InterruptedException e) {}

    }
}
```



## Работа с памятью

Основные операции, доступные для потоков при работе с памятью:

- **use** — чтение значения переменной из рабочей памяти потока;
- **assign** — запись значения переменной в рабочую память потока;
- **read** — получение значения переменной из основного хранилища;
- **load** — сохранение значения переменной, прочитанного из основного хранилища, в рабочей памяти;
- **store** — передача значения переменной из рабочей памяти в основное хранилище для дальнейшего хранения;
- **write** — сохраняет в основном хранилище значение переменной, переданной командой **store**.

Последовательность команд подчиняется следующим правилам:

- все действия, выполняемые одним потоком, строго упорядочены, т.е. выполняются одно за другим;
- все действия, выполняемые с одной переменной в основном хранилище памяти, строго упорядочены, т.е. следуют одно за другим.

# Volatile

Модификатор поля **volatile** устанавливает более строгие правила работы со значениями переменных, чтобы обеспечить всегда актуальное значение переменной и главного хранилища:

- перед **use** — всегда **load** и наоборот,
- перед **set** — всегда **assign** и наоборот,
- порядок работы с несколькими **volatile** переменными сохраняется и в передаче изменений в главное хранилище.

# Блокировки

Операции блокировки объекта в хранилище:

- **lock** — установить блокировку, только один поток в один момент времени может установить блокировку на некоторый объект;
- **unlock** — снять блокировку, если до того, как этот поток выполнит операцию `unlock`, другой поток попытается установить блокировку, его выполнение будет приостановлено до тех пор, пока первый поток не отпустит ее.

После успешно выполненного `lock` рабочая память очищается и все переменные необходимо заново считывать из основного хранилища. Аналогично, перед операцией `unlock` необходимо все переменные сохранить в основном хранилище.

# Synchronized

```
public class ThreadTest implements Runnable {
    private static ThreadTest shared = new ThreadTest();

    public void process() {
        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName() + " " + i);
            Thread.yield();
        }
    }

    public void run() {
        shared.process();
    }

    public static void main(String s[]) {
        for (int i = 0; i < 3; i++) {
            new Thread(new ThreadTest(), "Thread-" + i).start();
        }
    }
}
```

# Synchronized

```
public void run() {  
    synchronized (shared) {  
        shared.process();  
    }  
}
```

Thread-0 0

Thread-0 1

Thread-0 2

Thread-1 0

Thread-1 1

Thread-1 2

Thread-2 0

Thread-2 1

Thread-2 2

# Deadlock

```
public class DeadlockDemo {  
    public final static Object one = new Object(), two = new  
Object();  
    public static void main(String s[]) {  
        Thread t1 = new Thread() {\\...};  
        Thread t2 = new Thread() {\\...};  
  
        t1.start();  
        t2.start();  
    }  
}
```

# Deadlock

```
Thread t1 = new Thread() {  
    public void run() {  
        synchronized (one) {  
            Thread.yield();  
            synchronized (two) {  
                System.out.println("Success!");  
            }  
        }  
    }  
};
```

# Deadlock

```
Thread t2 = new Thread() {  
    public void run() {  
        synchronized (two) {  
            Thread.yield();  
            synchronized (one) {  
                System.out.println("Success!");  
            }  
        }  
    }  
};
```



# Wait-set

Каждый объект в Java имеет wait-set — набор потоков исполнения:

- Любой поток может вызвать метод **wait()** любого объекта и таким образом попасть в его wait-set.
- Выполнение такого потока приостанавливается до тех пор, пока другой поток не вызовет у этого же объекта метод **notifyAll()**, который пробуждает все потоки из wait-set.
- Метод **notify()** пробуждает один случайно выбранный поток из данного набора.

Любой из них может быть вызван потоком у объекта только после установления блокировки на этот объект. То есть либо внутри `synchronized`-блока с ссылкой на этот объект в качестве аргумента, либо обращения к методам должны быть в синхронизированных методах класса самого объекта.

## Пример wait-set

```
public class WaitThread implements Runnable {
    private Object shared;
    public WaitThread(Object o) {
        shared = o;
    }

    public void run() {
        synchronized (shared) {
            try {
                shared.wait();
            } catch (InterruptedException e) {
            }
            System.out.println("after wait");
        }
    }
    //...
}
```

## Пример wait-set

```
public static void main(String s[]) {  
    Object o = new Object();  
    WaitThread w = new WaitThread(o);  
    new Thread(w).start();  
    try {  
        Thread.sleep(100);  
    } catch (InterruptedException e) {  
    }  
    System.out.println("before notify");  
    synchronized (o) {  
        o.notifyAll();  
    }  
}
```

# Интерфейсы коллекций

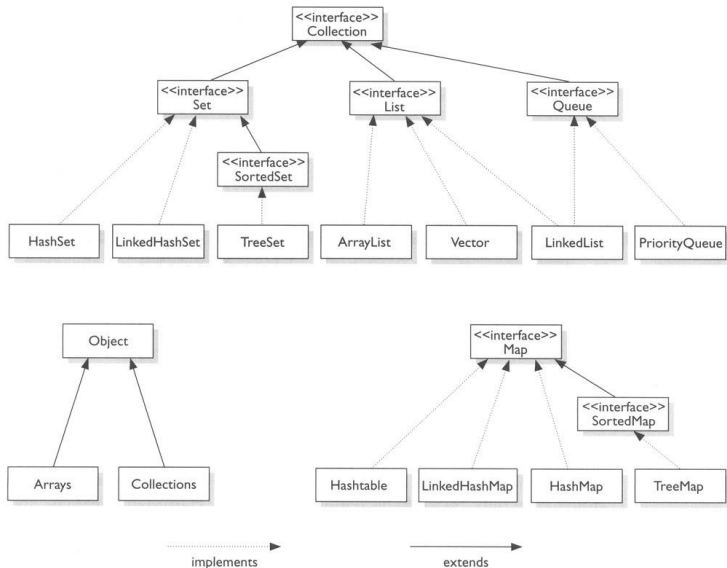
Все классы коллекций унаследованы от различных интерфейсов, которые определяют поведение коллекции.

Иными словами интерфейс определяет «что делает коллекция», а конкретная реализация «как коллекция делает то что определяет интерфейс».

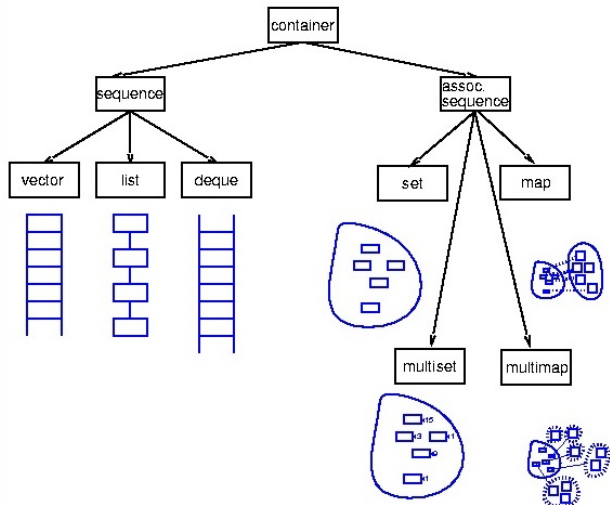
При разработке приложений рекомендуется, там где это возможного, использовать интерфейсы, т. к. это:

- позволяет легко заменять реализацию интерфейса, с целью повышения производительности, например,
- позволяет разработчику сконцентрироваться на задаче, а не на особенностях реализации.

# Обзор коллекций в Java



# Обзор коллекций в C++



## Пример использования коллекций

```
import java.util.*;
public class CollectionTest {
    public static void main(String[] args){
        List<String> list = new LinkedList<String>();
        list.add("bgsd");
        list.add("asdf");
        Collections.sort(list);

        for(String element : list){
            System.out.println(element);
        }

        System.out.println(list.get(1));
        System.out.println(list.indexOf("asdf"));
        System.out.println(list.contains("acf"));
        System.out.println(list.size());
        System.out.println(list.remove(0));
    }
}
```

## Пример использования коллекций

```
#include <string.h>
#include <algo.h>
#include <vector.h>
#include <stdlib.h>
#include <iostream.h>

main ()
{
    std::vector<int> v;
    int input;
    while (std::cin >> input)
        v.push_back (input);

    std::sort(v.begin(), v.end());

    int n = v.size();
    for (int i = 0; i < n; i++)
        std::cout << v[i] << "\n";
}
```



# Сборка на C++

Cmake — кроссплатформенный инструмент генерации файлов сборки под разные системы (make, VS, XCode, QtCreator и т. д.)

(<http://www.cmake.org/>, крайняя версия - 3.2.1).

- Создаем файл CMakeLists.txt с описанием проекта на языке cmake.
- Команда `cmake -G <make-system>` генерирует файл сборки.
- Открываем сгенерированный файл сборки в IDE или собираем его из командной строки (`make`, `nmake` и т. п.).

```
cmake_minimum_required (VERSION 2.6)
set (PROJECT hello_world)
project (${PROJECT})
set (HEADERS
    hello.h)
set (SOURCES
    hello.cpp
    main.cpp)
add_executable (${PROJECT} ${HEADERS} ${SOURCES})
```

# Сборка на C++

Подключение библиотеки в stake:

```
include_directories (../)
set (LIBRARIES
    lib_1
    lib_2)

foreach (LIBRARY ${LIBRARIES})
    add_subdirectory (../${LIBRARY}/build bin/${LIBRARY})
endforeach ()

target_link_libraries (${PROJECT} ${LIBRARIES})
```

# Сборка на Java

Maven - кроссплатформенная система сборки проектов (<http://www.maven.apache.org>, крайняя версия 3.2).

- Создаем файл `pom.xml` с кратким описанием проекта.
- Запускаем команду `mvn clean install` для сборки проекта в `.jar` или `.war` в зависимости от типа проекта.
- запускаем проект `java -jar <project>.jar`

```
<?xml version="1.0" encoding="UTF-8"?>  
<project ...>
```

```
<groupId>ru.isa.ai</groupId>  
<artifactId>ai-main</artifactId>  
<version>1.0</version>  
<packaging>jar</packaging>  
<name>ai-main</name>
```

```
<modules>  
  <module>ai-linguistic-utils</module>  
</modules>
```

# Сборка на Java

<http://mvnrepository.com> — сайт с большинством библиотек на Java.

```
<dependencies>
  <!-- Logging -->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.6.4</version>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.16</version>
  </dependency>
</dependencies>
```

# Сборка на Python

Файл `setup.py` - производится настройка и установка необходимых библиотек средствами самого языка `python`.

```
def printOptions(optionsDesc)  
def getVersion()  
def findRequirements()
```