КЛАССЫ И НАСЛЕДОВАНИЕ В JAVA

Основы наследования

- Наследование позволяет создавать иерархические классификации.
- Используя наследование, можно создать главный класс, который определяет свойства, общие для набора связанных элементов.
- Класс, который унаследован, называется *суперклассом* (superclass).
- Класс, который выполняет наследование, называется *подклассом* (subclass) — это специализированная версия суперкласса. Он наследует все переменные экземпляра и методы, определенные суперклассом, и прибавляет свои собственные уникальные элементы.

Основы наследования

- Чтобы наследовать класс, нужно просто включить определение одного класса в другое, используя ключевое слово extends.

Простой пример наследования

```
// Создать суперкласс,
class A { int i, j;
void showi j () {.
System.out.println("i и j: " + i + " " + j); } }
// Создать подкласс расширением класса A.
class B extends A { int k;
void showk() {
System.out.println("к: " + к); } void sum() {
System.out.println("i+j+k: " + (i+j+k)); } }
class SimpleInheritance {
public static void main(String args[])
 { A superOb = new A(); B subOb = new B();
// Суперкласс может быть использован сам
    по себе.
superOb.i = 10;
superOb.j = 20;
System.out.println("Содержимое superOb: ");
superOb.showij();
System.out.println();
/* Подкласс имеет доступ ко всем public-
    членам его суперкласса. */
subOb.i = 7;
subOb.j =8;       ,
subOb.k = 9;
System.out.println("Содержимое of subOb: ");
    subOb.showij(); subOb.showk();
    System.out.println();
System.out.println("Сумма i, j и к в subOb:");
    subOb.sum(); }
}
```

Доступ к элементам и наследование

- Хотя подкласс включает все элементы (члены) своего суперкласса, он не может обращаться к тем элементам суперкласса, которые были объявлены как *private*.

Создание многоуровневой иерархии

Мы можем сделать описание своих домашних животных (pets): кошек (cats), собак (dogs), коров (cows) и прочих следующим образом:

```
class Pet{     // Здесь описываем общие свойства всех домашних
    любимцев
    Master person; // Хозяин животного
     int weight, age, eatTimel];            // Вес, возраст, время кормления
     int eat(int food, int drink, int time){ // Процесс кормления
                                          // Начальные действия...
     if (time == eatTimefi]) person.getFood(food, drink);
                        // Метод потребления пищи
}
void voice();  // Звуки, издаваемые животным
                // Прочее...
}
```

Создание многоуровневой иерархии

Затем создаем классы, описывающие более конкретные объекты, связывая их с общим классом:

```
class Cat extends Pet{  // Описываются свойства,
   присущие только кошкам:
int mouseCatched;       // число пойманных мышей
void toMouse();         // процесс ловли мышей
                        // Прочие свойства
}

class Dog extends Pet{  // Свойства собак:
void preserve();        // охранять
}
```

Создание многоуровневой иерархии

Заметьте, что мы не повторяем общие свойства, описанные в классе Pet . Они наследуются автоматически. Мы можем определить объект класса Dog и использовать в нем все свойства класса Pet так, как будто они описаны в классе Dog :

```
Dog tuzik = new Dog(), sharik = new Dog();
```

После этого определения можно будет написать

```
tuzik.age = 3;
int p = sharik.eat (30, 10, 12);
```

А классификацию продолжить так:

```
class Pointer extends Dog{ ... } // Свойства породы Пойнтер
class Setter extends Dog{ ... }  // Свойства сеттеров
```

Создание многоуровневой иерархии

Заметьте, что на каждом следующем уровне иерархии в класс добавляются новые свойства, но ни одно свойство не пропадает. Поэтому и употребляется слово extends — "расширяет" и говорят, что класс Dog — расширение (extension) класса Pet . С другой стороны, количество объектов при этом уменьшается: собак меньше, чем всех домашних животных. Поэтому часто говорят, что класс Dog — подкласс (subclass) класса Pet , а класс Pet — суперкласс (superclass) или надкласс класса Dog .

Часто используют генеалогическую терминологию: родительский класс, дочерний класс, класс-потомок, класс-предок. Класс Dog наследует класс Pet .

Опишем в классе Master владельца домашнего зоопарка.

```
class Master{ // Хозяин животного
String name;  // Фамилия, имя
        // Другие сведения
void getFood(int food, int drink); // Кормление
        // Прочее
}
```

Модификаторы ограничения доступа к элементам при наследовании

```
class Bisection2{
private static double final EPS = 1e-8; // Константа
private double a = 0.0, b = 1.5, root;  // Закрытые поля
public double getRoot(){return root;}   // Метод доступа
private double f(double x)
{
return x*x*x — 3*x*x + 3;   // Или что-то другое
}
private void bisect(){      // Параметров нет —
                    // метод работает с полями экземпляра
double y = 0.0;   // Локальная переменная — не поле
do{
root = 0.5 *(a + b); y = f(root);
if (Math.abs(y) < EPS) break;
// Корень найден. Выходим из цикла
// Если на концах отрезка [a; root]
// функция имеет разные знаки:
if (f(a) * y < 0.0} b = root;
     // значит, корень здесь
     // Переносим точку b в точку root
     //В противном случае:
else a = root;
     // переносим точку a в точку root
     // Продолжаем, пока [a; Ь] не станет мал
} while(Math.abs(b-a) >= EPS);
}

public static void main(String[] args){
Bisection2 b2 = new Bisection2();
b2.bisect();
System.out.println("x = " +
b2.getRoot() +    // Обращаемся к корню
       // через метод доступа
", f() = " +b2.f(b2.getRoot()));
}
}
```

Понятие и использование абстрактных классов

- При описании класса Pet мы не можем задать в методе voice () никакой полезный алгоритм, поскольку у всех животных совершенно разные голоса. В таких случаях мы записываем только заголовок метода и ставим после закрывающей список параметров скобки точку с запятой. Этот метод будет абстрактным (abstract).
- Использовать абстрактные классы можно только порождая от них подклассы, в которых переопределены абстрактные методы.

Понятие и использование абстрактных классов

- Хотя элементы массива singer [] ссылаются на подклассы Dog, Cat, Cow, но все-таки это переменные типа Pet и ссылаться они могут только на поля и методы, описанные в суперклассе Pet . Дополнительные поля подкласса для них недоступны. Если обратиться, например, к полю k класса Dog , написав singer [0].k , то получим отклик о невозможности реализовать такую ссылку. Поэтому метод, который реализуется в нескольких подклассах, приходится выносить в суперкласс, а если там его нельзя реализовать, то объявить абстрактным. Абстрактные классы группируются на вершине иерархии классов.
- Можно задать пустую реализацию метода, просто поставив пару фигурных скобок, ничего не написав между ними, например: void voice(){}
- Получится полноценный метод. Но это искусственное решение, запутывающее структуру класса.

Окончательные члены и классы

- Пометив метод модификатором final , можно запретить его переопределение в подклассах. Это удобно в целях безопасности. Вы можете быть уверены, что метод выполняет те действия, которые вы задали. Именно так определены математические функции sin(), cos() и прочие в классе Math . Мы уверены, что метод Math.cos (x) вычисляет именно косинус числа x . Разумеется, такой метод не может быть абстрактным.
- Для полной безопасности, поля, обрабатываемые окончательными методами, следует сделать закрытыми (private).

Окончательные члены и классы

- Если же пометить модификатором final весь класс, то его вообще нельзя будет расширить. Так определен, например, класс Math :

```
public final class Math{ . . . }
```

- Для переменных модификатор final имеет совершенно другой смысл. Если пометить модификатором final описание переменной, то ее значение (а оно должно быть обязательно задано или здесь же, или в блоке инициализации или в конструкторе) нельзя изменить ни в подклассах, ни в самом классе. Переменная превращается в константу. Именно так в языке Java определяются константы:

```
public final int MIN_VALUE = -1, MAX_VALUE = 9999;
```

- По соглашению "Code Conventions" константы записываются прописными буквами, слова в них разделяются знаком подчеркивания.
- На самой вершине иерархии классов Java стоит класс Object .

Класс Object

- В Java определен один специальный класс — object. Все другие классы являются его подклассами, object — это суперкласс всех других классов. Это означает, что ссылочная переменная типа object может обращаться к объекту любого другого класса. Кроме того, т. к. массивы реализуются как классы, переменная типа object может также обращаться к любому массиву.

Класс Object

- Если при описании класса мы не не пишем слово extends и имя класса за ним, то Java считает этот класс расширением класса object , и компилятор дописывает это за нас:

```
class Pet extends Object{ . . . }
```

- Можно записать это расширение и явно.
- Сам же класс object не является ничьим наследником, от него начинается иерархия любых классов Java. В частности, все массивы — прямые наследники класса object .
- Поскольку такой класс может содержать только общие свойства всех классов, в него включено лишь несколько самых общих методов, например, метод equals() , сравнивающий данный объект на равенство с объектом, заданным в аргументе, и возвращающий логическое значение. Его можно использовать так:

```
Object obj1 = new Dog(), obj2 = new Cat();
if (obj1.equals(obj2)) ...
```

Класс Object

- Объект obj1 активен, он сам сравнивает себя с другим объектом. Можно, конечно, записать и obj2.equals (obj1) , сделав активным объект obj2 , с тем же результатом.
- Ссылки можно сравнивать на равенство и неравенство:

obj1 == obj2; obj1 != obj 2;

- В этом случае сопоставляются адреса объектов, мы можем узнать, не указывают ли обе ссылки на один и тот же объект.

Класс Object

- Метод equals() же сравнивает содержимое объектов в их текущем состоянии, фактически он реализован в классе object как тождество: объект равен только самому себе. Поэтому его часто переопределяют в подклассах, более того, правильно спроектированные классы должны переопределить методы класса object , если их не устраивает стандартная реализация.
- Второй метод класса object , который следует переопределять в подклассах, — метод tostring () . Это метод без параметров, который пытается содержимое объекта преобразовать в строку символов и возвращает объект класса string .
- К этому методу исполняющая система Java обращается каждый раз, когда требуется представить объект в виде строки, например, в методе printing .

Методы object

Метод	Цель
Object clone ()	Создает новый объект, который является таким же, как имитируемый объект
boolean equals (Object object)	Определяет, является ли один объект равным другому
void finalize ()	Вызывается прежде, чем неиспользованный объект будет переработан (сборщиком мусора)
Class getciass ()	Получает класс объекта во время выполнения
int hashCode()	Возвращает хэш-код, связанный с вызовом объекта
void notify()	Возобновляет выполнение потока, ожидающего на объекте вызова
void notifyAU()	Возобновляет выполнение всех потоков, ожидающих на объекте вызова
string toStringO	Возвращает строку, которая описывает объект
void wait () void wait(long milllsrconds) void wait(long mllllsrconds, int nanoseconds)	Ждет выполнения на другом потоке

Методы object

- Методы getClass(), notify (), notifyAU() и wait () объявлены как final. Другие можно переопределять.
- Здесь отметим два метода: equals() и toString().
- Метод equals() о сравнивает содержимое двух объектов. Он возвращает true, если объекты эквивалентны, и false— в противном случае.
- Метод ToString() возвращает строку, содержащую описание объекта, на котором он вызывается. Кроме того, этот метод вызывается автоматически, когда объект выводится методом println ().

Использование ключевого слова super (первый вид)

- Подкласс может вызывать метод конструктора, определенный его суперклассом, при помощи следующей формы super:

super (parameter—*list) ;*

- Здесь **parameter-list** — список параметров, который определяет любые параметры, необходимые конструктору в суперклассе. Похожий по форме на конструктор super () должен всегда быть первым оператором, выполняемым внутри конструктора подкласса.

```
// BoxWeight теперь использует
// super для инициализации
// Box-атрибутов.
class BoxWeight extends Box {
double weight; // вес блока
// инициализировать width,
// height и depth, используя
// super()
BoxWeight(double w, double h,
    double d, double m) {
super(w, h, d); // вызвать
// конструктор суперкласса
weight = m; } }
```

Использование второй формы *super*

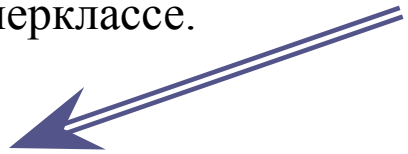
- Общий формат такого использования super имеет вид:

super. *member*

- где *member* может быть либо методом, либо переменной экземпляра.
- Вторая форма super больше всего применима к ситуациям, когда имена элементов (членов) подкласса скрывают элементы с тем же именем в суперклассе.

```
// Использование super для
// преодоления скрытия имен,
class A {int i ; }
// Создание подкласса B расширением
// класса A.
class B extends A {
int i;  // этот i скрывает i в A
B(int a, int b) {
super.i = a; // i из A
i = b;// i из B
}
void show() {
System.out.println("i из суперкласса: " + super.i) ;
System.out.println("i из подкласса: " + i); } }
class UseSuper {

public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show() ; }
}
```



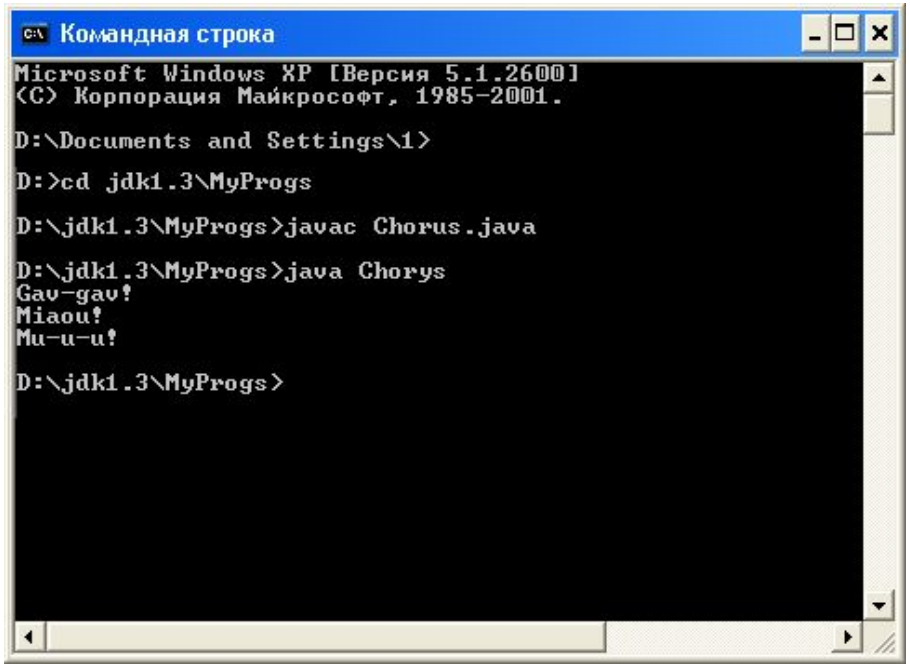
```
i из суперкласса:   1
i из подкласса:   2
```

Переопределение методов

```
abstract class Pet{
   abstract void voice();  }
class Dog extends Pet{
void voice(){
      System.out.printin("Gav-gav!");    } }
class Cat extends Pet{    void voice () {
     System.out.printin("Miaou!");     } }
class Cow extends Pet{     void voice(){
      System.out.printin("Mu-u-u!");    } }

public class Chorus(
   public static void main(String[] args){
      Pet[] singer = new Pet[3];
      singer[0] = new Dog();
      singer[1] = new Cat();
      singer[2] = new Cow();
      for (int i = 0; i < singer.length; i++)
         singer[i].voice();
   }
}
```

Все дело здесь в определении поля singer[]. Хотя массив ссылок singer [] имеет тип Pet , каждый его элемент ссылается на объект своего типа Dog, Cat, cow . При выполнении программы вызывается метод конкретного объекта, а не метод класса, которым определялось имя ссылки.



Переопределение методов

```
class A { int i,   j;
A(int a,   int b)   {
i  = a;
j  = b; }
// показать i и j на экране
void show() (
System.out.println("i и j: " + i + " " + j); } }
class B extends A { int k;
B(int a, int b, int c) {
super(a, b);
k = c; }
// Показать на экране к (этот show(i переопределяет show() из A)
void show() {
System.out.println("k: " + k); } }
class Override {
public static void main(String args[]) { B subOb = new B(1, 2, 3);
subOb.showO;  // здесь вызывается show() из B
} }
```

Динамическая диспетчеризация методов

- Динамическая диспетчеризация методов это механизм, с помощью которого решение на вызов переопределенной функции принимается во время выполнения, а не во время компиляции.
- Принцип: ссылочная переменная суперкласса может обращаться к объекту подкласса. Java использует этот факт, чтобы принимать решения о вызове переопределенных методов во время выполнения.

Динамическая диспетчеризация методов

- Когда переопределенный метод вызывается через ссылку суперкласса, Java определяет, какую версию этого метода следует выполнять, основываясь на *типе объекта,* на который указывает ссылка в момент вызова. Это определение делается *во время выполнения.* Когда ссылка указывает на различные типы объектов, будут вызываться различные версии переопределенного метода.
- Другими словами, именно *тип объекта, на который сделана ссылка* (а не тип ссылочной переменной) определяет, какая версия переопределенного метода будет выполнена.

Динамическая диспетчеризация методов

```
class A {
void callme ()    {
System.out.println("Внутри A метод
    callme"); } }
class B extends A (
// переопределить callme()
void callme{) {
System.out.println("Внутри B метод
    callme");
} }
class C extends A {
// переопределить callme()
void callme () {
System.out.println("Внутри C метод
    callme");
}}
```

```
class Dispatch {
public static void main(String args[]) {
    A a = new A();    // объект типа A
B b = new B();        // объект типа B
C c = new C();         // объект типа C
    A r;    // определить ссылку типа A
r = a;      // r на A-объект
r.callme();// вызывает A-версию callme
r = b;      // r указывает на B-объект
r.callme();// вызывает B-версию callme
r = c;      //r указывает,на C-объект       ,
r.callme();// вызывает C-версию callme
}
}
```

Использование ключевого слова *final* с наследованием

- Ключевое слово final имеет три применения. Первое — его можно использовать для создания эквивалента именованной константы. Два других применения final связаны с наследованием.

Использование *final* для отказа от переопределения

- Чтобы отменить переопределение метода, укажите модификатор final в начале его объявления. Методы, объявленные как final, не могут переопределяться.

```
class A  {
final void meth() {
System.out.println("Это метод final.");
} }
class B extends A {
void meth() {  // ОШИБКА! Нельзя переопределять.
System.out.println("Ошибка!");
} }
```

- Поскольку meth о объявлен как final, он не может быть переопределен в классе B. Если вы попытаетесь сделать это, то получите ошибку во время компиляции.

Использование *final* для отмены наследования

- Иногда нужно разорвать наследственную связь классов (отменить наследование одного класса другим). Чтобы сделать это, предварите объявление класса ключевым словом final, что позволит неявно объявить и все его методы. Недопустимо объявлять класс одновременно как abstract и final.

```
final class A { // ...
}
// Следующий класс незаконный.
class B extends A {        //ОШИБКА! B не может быть подклассом A
// .. . }
```

- Комментарий здесь означает, что в не может наследовать А, т.к. А объявлен как final.

Спасибо за внимание!