Определение классов (продолжение)

Оглавление

Перегруженные методы

Перегруженными (overloaded) методами называются методы одного класса с одинаковыми именами. Сигнатуры у них должны быть различными и различие может быть только в наборе аргументов.

Если в классе параметры перегруженных методов заметно различаются: например, у одного метода один параметр, у другого – два, то для Java это совершенно независимые методы и совпадение их имен может служить только для повышения наглядности работы класса. Каждый вызов, в зависимости от количества параметров, однозначно адресуется тому или иному методу.

Однако если количество параметров одинаковое, а типы их различаются незначительно, при вызове может сложиться двойственная ситуация, когда несколько перегруженных методов одинаково хорошо подходят для использования. Например, если объявлены типы Parent и Child, где Child расширяет Parent, то для следующих двух методов:

```
void process(Parent p, Child c) {}
void process(Child c, Parent p) {}
```

можно сказать, что они допустимы, их сигнатуры различаются. Однако при вызове

```
process(new Child(), new Child());
```

обнаруживается, что оба метода одинаково годятся для использования. Другой пример, методы:

```
process(Object o) {}
process(String s) {}
```

и примеры вызовов:

```
process(new Object());
process(new Point(4,5));
process("abc");
```

Очевидно, что для первых двух вызовов подходит только первый метод, и именно он будет вызван. Для последнего же вызова подходят оба перегруженных метода, однако класс String является более "специфичным", или узким, чем класс Object. Действительно, значения типа String можно передавать в качестве аргументов типа Object, обратное же неверно. Компилятор попытается отыскать наиболее специфичный метод, подходящий для указанных параметров, и вызовет именно его. Поэтому при третьем вызове будет использован второй метод.

Однако для предыдущего примера такой подход не дает однозначного ответа. Оба метода одинаково специфичны для указанного вызова, поэтому возникнет ошибка компиляции. Необходимо, используя явное приведение, указать компилятору, какой метод следует применить:

```
process((Parent)(new Child()), new Child());
// или
process(new Child(),(Parent)(new Child()));
```

Это верно и в случае использования значения null:

```
process((Parent)null, null);
// или
process(null,(Parent)null);
```

Ключевое слово this

Ключевое слово this является стандартной ссылкой на объект, из которого вызывается метод. При этом следует учесть, что в Java ссылка на объект фактически является именем этого объекта. Хотя наличие такой ссылки может на первый взгляд показаться излишним, она достаточно часто используется на практике.

Условно можно выделить два типа ситуаций, когда может потребоваться ссылка **this**: если она реально необходима и, если благодаря ей улучшается читабельность программного кода. Для их понимания необходимо сначала познакомиться с особенностями выполнения ссылок на объекты, способами передачи аргументов методам и механизмом возвращения методом объекта в качестве результата. Приведем простые примеры использования ключевого слова **this**.

В рассматривавшихся ранее примерах ссылки на члены класса в теле методов этого же класса выполнялись простым указанием имени соответствующего члена.

Если для класса создается объект и из этого объекта вызывается метод, то обращение к члену класса в программном коде метода означает обращение к соответствующему члену объекта. В то же время обращение к полям и методам объекта выполняется в «точечном» синтаксисе, то есть указывается имя объекта, точка и затем имя поля или метода. Для упомянутой ситуации это означает, что в методе объекта выполняется обращение к члену того же объекта. Этот факт можно отразить в программном коде метода в явном виде с помощью ссылки this. Пример приведен в листинге:

```
class MyClass{
// Поля класса:
double Re,Im;
void set(double Re,double Im){
// Использование ссылки this:
this.Re=Re;
this.Im=Im;}
void get(){
// Инструкция перехода на новую строку \n:
System.out.println("Значения полей:\nRe="+this.Re+" и Im="+this.Im);}
}
class ThisDemo{
public static void main(String[] args){
MyClass obj=new MyClass();
obj.set(1,5);
obj.get();}
}
```

В данном случае класс MyClass содержит всего два поля Re и Im типа double. Кстати, обращаем внимание, что оба поля объявлены одной инструкцией, в которой указан тип полей, а сами поля перечислены через запятую. Данный класс является очень слабой аналогией реализации комплексных чисел, у которых есть действительная (поле Re) и мнимая (поле Im) части.

Кроме двух полей, у класса имеется два метода: метод set() для присваивания значений полям и метод get() для отображения значений полей. У метода set() два аргумента типа double, причем их названия совпадают с названиями полей класса. Разумеется, в таком экстремизме необходимости нет, вполне можно было предложить иные названия для аргументов, но мы не ищем простых путей, поэтому сложившаяся ситуация далеко не однозначна. Если в теле метода set() использовать обращение Re или Im, то это будет ссылка на аргумент метода, а не на поле класса, поскольку в случае совпадения имен переменных приоритет имеет локальная переменная (то есть объявленная в блоке, в котором находится инструкция вызова переменных с совпадающими именами). Из ситуации выходим, воспользовавшись ссылкой this. Так, инструкции this. Re и this.Im означают поле Re и поле Im соответственно объекта this, то есть того объекта, из которого вызывается метод set(). При этом инструкции Re и Im являются обращениями к аргументам метода. Поэтому, например, команду this.Re=Re следует понимать так: полю Re объекта this (объекта, из которого вызывается метод set()) присвоить значение аргумента Re, то есть первого аргумента, переданного методу set() при вызове. Хотя описанная ситуация вполне оправдывает использование ссылки this, все же такой прием считается несколько искусственным, поскольку в запасе всегда остается возможность просто поменять названия аргументов.

Совсем неоправданным представляется использование ссылки this в программном коде метода get(), который выводит сообщение со значениями полей объекта. В данном случае инструкции вида this.Re и this.Im можно заменить простыми обращениями Re и Im соответственно — функциональность кода не изменится. У метода аргументов нет вообще, поэтому никаких неоднозначностей не возникает. По большому счету, обращение к полю класса (или методу) по имени является упрощенной формой ссылки this (без применения самого ключевого слова this). Этой особенностью синтаксиса языка Java мы пользовались ранее и будем пользоваться в дальнейшем. Тем не менее в некоторых случаях указание ссылки this даже в «косметических» целях представляется оправданным.

Обращаем внимание на инструкцию перехода на новую строку \n в текстовом аргументе метода println() в теле метода get(). Эта инструкция включена непосредственно в текст и ее наличие приводит к тому, что в месте размещения инструкции при выводе текста выполняется переход на новую строку. Поэтому в результате выполнения программы получим сообщение из двух строк:

Значения полей:
Re=1.0 и Im=5.0

Внутренние классы

Внутренний класс — это класс, объявленный внутри другого класса. Эту ситуацию не следует путать с использованием в качестве поля класса объекта другого класса. Здесь речь идет о том, что в рамках кода тела класса содержится описание другого класса, который и называется внутренним. Класс, в котором объявлен внутренний класс, называется внешним. В принципе, внутренний класс может быть статическим, но такие классы используются на практике крайне редко, поэтому рассматривать мы их не будем, а ограничимся только нестатическими внутренними классами.

Внутренний класс имеет несколько особенностей. Во-первых, члены внутреннего класса доступны только в пределах внутреннего класса и недоступны во внешнем классе (даже если они открытые). Во-вторых, во внутреннем классе можно обращаться к членам внешнего класса напрямую. Наконец, объявлять внутренние классы можно в любом блоке внешнего класса. Пример использования внутреннего класса приведен в листинге:

```
class MyOuter{
// Поле внешнего класса:
int number=123;
// Метод внешнего класса:
void show(){
// Создание объекта внутреннего класса:
MyInner InnerObj=new MyInner();
// Вызов метода объекта внутреннего класса:
InnerObj.display();}
// Внутренний класс:
class MyInner{
// Метод внутреннего класса:
void display(){
System.out.println("Поле number="+number);}
```

```
}
}
class InnerDemo{
public static void main(String args[]){
// Создание объекта внешнего класса:
MyOuter OuterObj=new MyOuter();
// Вызов метода объекта внешнего класса:
OuterObj.show();}
}
```

В программе описаны три класса: внешний класс MyOuter, описанный в нем внутренний класс MyInner, а также класс InnerDemo. В классе InnerDemo описан метод main(), в котором создается объект внешнего класса MyOuter и вызывается метод этого класса show().

Структура программы следующая: во внешнем классе MyOuter объявляется поле number, метод show() и описывается внутренний класс MyInner. У внутреннего класса есть метод display(), который вызывается из метода внешнего класса show(). Для вызова метода display() в методе show() создается объект внутреннего класса InnerObj. Причина в том, что вызывать метод display() напрямую нельзя — члены внутреннего класса во внешнем классе недоступны.

В методе display() выводится сообщение со значением поля внешнего класса number. Поскольку во внутреннем классе допускается непосредственное обращение к членам внешнего класса, обращение к полю number выполняется простым указанием его имени.

В результате выполнения программы получаем сообщение:

Поле number=123

Отметим, что в главном методе программы можно создать объект внешнего класса, но нельзя создать объект внутреннего класса — за пределами внешнего класса внутренний класс недоступен.

Анонимные объекты

Как уже отмечалось, при создании объектов с помощью оператора new возвращается ссылка на вновь созданный объект. Прелесть ситуации состоит в том, что эту ссылку не обязательно присваивать в качестве значения переменной. В таких случаях создается анонимный объект. Другими словами, объект есть, а переменной, которая бы содержала ссылку на этот объект, нет. С практической точки зрения такая возможность представляется сомнительной, но это только на первый взгляд.

На самом деле анонимные объекты требуются довольно часто — обычно в тех ситуациях, когда единожды используется единственный объект класса. Достаточно простой пример применения анонимного объекта приведен в листинге:

```
class MyClass{
void show(String msg){
System.out.println(msg);}
}
class NamelessDemo{
public static void main(String args[]){
// Использование анонимного объекта:
new MyClass().show("Этот объект не имеет имени");}
```

```
}
```

В классе MyClass описан всего один метод show() с текстовым аргументом. Текстовый аргумент — это объект встроенного Java-класса String. Действие метода состоит в том, что на экран выводится текст, переданный аргументом метода.

В методе main() класса NamelessDemo всего одна команда, которая и демонстрирует применение анонимного объекта:

```
new MyClass().show("Этот объект не имеет имени")
```

Эту команду можно условно разбить на две части. Инструкцией new MyClass() создается новый объект класса MyClass, а сама инструкция в качестве значения возвращает ссылку на созданный объект. Поскольку ссылка никакой переменной в качестве значения не присваивается, созданный объект является анонимным. Однако это все равно объект класса MyClass, поэтому у него есть метод show(). Именно этот метод вызывается с аргументом "Этот объект не имеет имени".

Для этого после инструкции new MyClass() ставится точка и имя метода с нужным аргументом.

Способы передачи аргументов

В Java существует два способа передачи аргументов методам: по значению и по ссылке. При передаче аргумента по значению в метод передается копия этого аргумента. Другими словами, если аргумент передается по значению, то перед передачей аргумента методу сначала создается безымянная копия переменной, передаваемой аргументом, и именно эта безымянная копия используется при вычислениях в рамках выполнения метода. После того как метод завершит свою работу, эта безымянная переменная автоматически уничтожается.

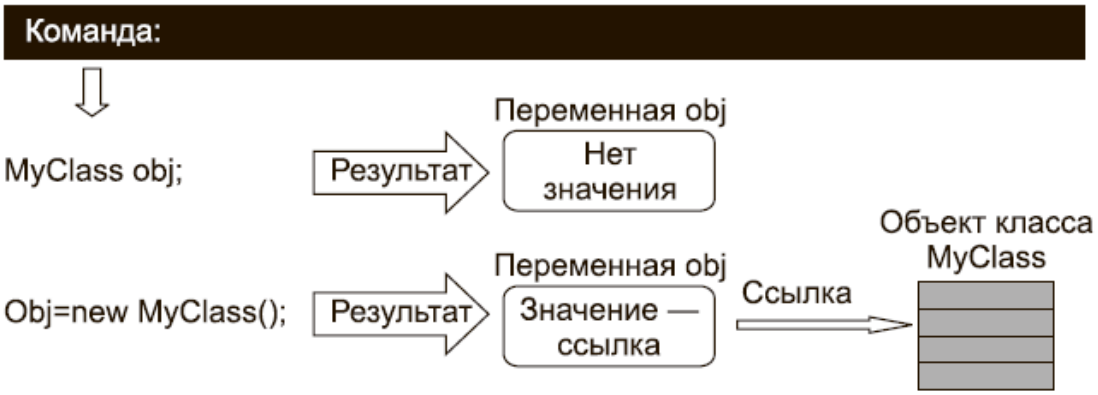
Если аргумент передается по ссылке, все операции в теле метода выполняются непосредственно с аргументом.

На практике разница между способами передачи аргументов проявляется в том случае, если в методе предпринимается попытка изменить аргументы метода.

В Java переменные базовых (простых) типов передаются по значению, а объекты — по ссылке. Хотя на первый взгляд это деление может показаться несколько искусственным, оно представляется практически необходимым, если принять во внимание способ создания объектов и реальную связь объекта и объектной переменной. Ранее этот механизм уже обсуждался, но нелишне будет напомнить еще раз.

Итак, при создании объектной переменной (это переменная типа «класс») объект не создается. Для этой переменной выделяется место в памяти, и в эту область памяти может быть записана ссылка на объект, не больше. Для создания объекта используется оператор new с вызовом конструктора для создания объекта. В этом случае в памяти выделяется место под объект, если конструктором предусмотрено, то заполняются поля этого объекта и выполняются нужные действия. Результатом, который может быть присвоен

объектной переменной, является ссылка на созданный объект. Стандартный процесс создания объекта иллюстрирует рис.



Когда объект передается методу в качестве аргумента, аргументом на самом деле указывается имя объекта, то есть ссылка на этот объект. В принципе, ссылка передается по значению, то есть создается анонимная копия объектной переменной. Это означает, что переменная имеет то же значение ссылки. Другими словами, анонимная копия ссылается на тот же объект, что и оригинальный аргумент-имя объекта. Операции над объектной переменной затрагивают объект, на который она ссылается. Поэтому изменения, применяемые к объекту, на самом деле изменяют сам объект.

В листинге приведен пример программы, в которой иллюстрируется разница в способах передачи аргументов методам:

```
class Base{
int a,b;
void show(){
System.out.println(a+":"+b);}
}

class Test{
void f(int x,int y){
x*=2;
y/=2;
System.out.println(x+":"+y);}
void f(Base obj){
obj.a*=2;
obj.b/=2;
System.out.println(obj.a+":"+obj.b);}
}

class TestDemo{
public static void main(String args[]){
Base obj=new Base();
Test tstFunc=new Test();
obj.a=10;
obj.b=200;
tstFunc.f(obj.a,obj.b);
obj.show();
tstFunc.f(obj);
obj.show();}
```

```
}
```

В классе Base имеются два целочисленных поля a и b, а также метод show(), предназначенный для отображения значений этих полей.

В классе Test описан перегруженный метод f(). У метода два варианта: с двумя аргументами типа int и с одним аргументом-объектом класса Base. В первом случае при выполнении метода f() его первый аргумент умножается на два, а второй делится на два. Затем выводится сообщение с парой значений первого и второго аргументов (после внесения в них изменений). Во втором варианте метода та же процедура проделывается с полями объекта, указанного аргументом метода:

первое поле объекта умножается на два, а второе делится на два, и новые значения полей выводятся в сообщении.

В главном методе программы в классе TestDemo создаются два объекта: объект obj класса Base и объект tstFunc класса Test. Командами obj.a=10 и obj.b=200 полям объекта obj присваиваются значения, после чего командой tstFunc.f(obj.a,obj.b) вызывается вариант метода f(), аргументами которого указаны поля объекта obj. В результате, как и следовало ожидать, появляется сообщение 20:100. Однако если проверить значения полей объекта obj с помощью команды obj.show(), то мы получим сообщение 10:200, означающее, что значения полей не изменились. Причина в том, что, хотя объект передается аргументом по ссылке, целочисленные поля объекта — это переменные базового типа, поэтому передаются по значению. По этой причине при вызове метода f() изменялись копии этих полей, а сами поля не изменились.

После выполнения команды tstFunc.f(obj) мы снова получаем сообщение 20:100.

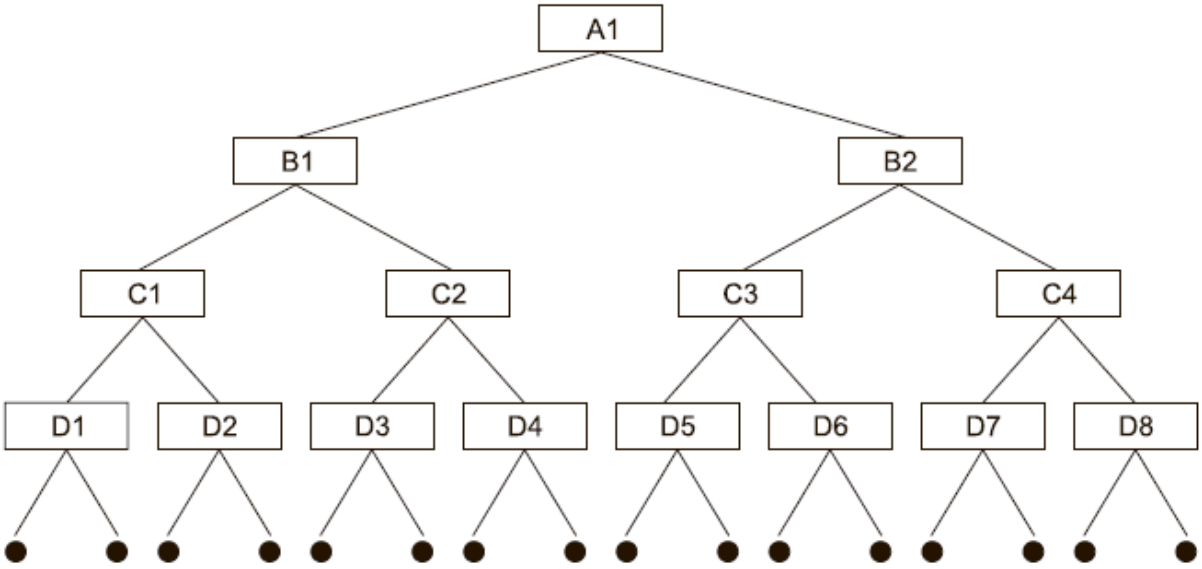
Такое же сообщение мы получаем при выполнении команды obj.show(). Следовательно, поля объекта изменились. В данном случае это вполне ожидаемо: аргументом методу передавался объект, а объекты передаются по ссылке, поэтому все вносимые изменения сказываются на самом объекте.

Теперь можно уточнить, что означает возможность объявлять параметры методов и конструкторов как final. Поскольку изменения значений параметров (но не объектов, на которые они ссылаются) никак не сказываются на переменных вне метода, модификатор final говорит лишь о том, что значение этого параметра не будет меняться на протяжении работы метода.

Бинарное дерево

Рассмотрим программу, в которой на основе конструкторов класса создается бинарное дерево объектов — каждый объект имеет по две ссылки на объекты того же класса. Пример учебный, поэтому ситуация упрощена до предела. Каждый объект, кроме прочего, имеет три поля. Символьное (типа char) поле Level определяет уровень объекта: например, в вершине иерархии находится объект уровня A, который ссылается на два объекта уровня B, которые, в свою очередь, ссылаются в общей сложности на четыре объекта уровня C и

т. д. Объекты нумеруются, для чего используется целочисленное поле Number. Нумерация выполняется в пределах одного уровня. Например, на верхнем уровне A всего один объект с номером 1. На втором уровне B два объекта с номерами 1 и 2. На третьем уровне C четыре объекта с номерами от 1 до 4 включительно и т. д. Описанная структура объектов представлена на рис.



Кроме метки уровня и номера объекта на уровне, каждый объект имеет еще и свой «идентификационный код». Этот код генерируется случайным образом при создании объекта и состоит по умолчанию из восьми цифр (количество цифр в коде определяется закрытым статическим целочисленным полем IDnum).

Что касается самого кода, то он записывается в целочисленный массив, на который ссылается переменная массива ID — закрытое поле класса ObjectTree. Каждая цифра кода записывается отдельным элементом соответствующего массива. Для генерирования (создания) кода объекта используется метод getID(). Для этого в теле метода командой ID[i]=(int)(Math.random()*10) рамках цикла генерируются случайные целые числа (инструкцией (int)(Math.random()*10)) и записываются в качестве значений элементов массива ID.

Для вывода кода объекта используется закрытый метод showID(). Методом последовательно выводятся на экран элементы массива ID, при этом в качестве разделителя используется вертикальная черта. Сам метод вызывается в методе show(), который, в свою очередь, предназначен для отображения параметров объекта: уровня объекта в структуре, порядкового номера объекта на уровне и идентификационного кода объекта.

Открытые поля FirstRef и SecondRef являются объектными переменными класса ObjectTree и предназначены для записи ссылок на объекты следующего уровня.

Присваивание значений этим переменным выполняется при вызове конструктора класса. Приведем соответствующий листинг:

```
//Бинарное дерево объектов
```

```
class ObjectTree{
// Количество цифр в ID-коде объекта:
private static int IDnum=8;
// Уровень объекта (буква):
private char Level;
// Номер объекта на уровне:
private int Number;
// Код объекта (массив цифр):
private int[] ID;
// Ссылка на первый объект:
ObjectTree FirstRef;
// Ссылка на второй объект:
ObjectTree SecondRef;
// Метод для генерирования ID-кода объекта:
private void getID(){
ID=new int[IDnum];
for(int i=0;i<IDnum;i++)
ID[i]=(int)(Math.random()*10);
}
// Метод для отображения ID-кода объекта:
private void showID(){
for(int i=0;i<IDnum;i++)
System.out.print("|"+ID[i]);
System.out.print("|\n");
}
// Метод для отображения параметров объекта:
void show(){
System.out.println("Уровень объекта: \t"+Level);
System.out.println("Номер на уровне: \t"+Number);
System.out.print("ID-код объекта: \t");
showID();
}
// Конструктор создания бинарного дерева:
ObjectTree(int k,char L,int n){
System.out.println("\tСоздан новый объект!");
Level=L;
Number=n;
getID();
show();
if(k==1){
FirstRef=null;
SecondRef=null;
}
else{
// Рекурсивный вызов конструктора:
FirstRef=new ObjectTree(k-1,(char)((int)L+1),2*n-1);
SecondRef=new ObjectTree(k-1,(char)((int)L+1),2*n);}
}}
class ObjectTreeDemo{
public static void main(String[] args){
// Дерево объектов:
ObjectTree tree=new ObjectTree(4,'A',1);
System.out.println("\tПроверка структуры дерева объектов!");
// Проверка структуры дерева объектов:
tree.FirstRef.SecondRef.FirstRef.show();
}}
```

Конструктор класса принимает три аргумента: целочисленный аргумент определяет количество уровней в структуре, начиная с текущего объекта, а символьные аргументы определяют метку уровня и номер объекта на уровне. Детальнее остановимся на коде конструктора, поскольку именно при его вызове создается вся структура бинарного дерева.

При вызове конструктора выводится сообщение о создании объекта, после чего на основе значений аргументов конструктора присваиваются значения полям Level и Number. Затем с помощью метода getID() генерируется идентификационный код объекта и методом show() выводится информация о созданном объекте. Вторая часть кода конструктора реализована через условную инструкцию.

В ней первый аргумент конструктора проверяется на предмет равенства единице.

Если первый аргумент равен единице (это означает, что после текущего объекта других объектов нет), полям-ссылкам FirstRef и SecondRef в качестве значений присваиваются нулевые ссылки (значение null), означающие, что текущий объект не имеет ссылок на другие объекты. В противном случае, то есть если аргумент конструктора отличен от единицы, следующими командами создаются два новых объекта, и ссылки на них в качестве значений присваиваются полям FirstRef и SecondRef текущего объекта:

```
FirstRef=new ObjectTree(k-1,(char)((int)L+1),2*n-1);
SecondRef=new ObjectTree(k-1,(char)((int)L+1),2*n);
```

При этом конструкторам при создании новых объектов передается на единицу уменьшенный первый аргумент. Уровень новых создаваемых объектов вычисляется на основе текущего значения L для уровня текущего объекта как (char)((int)L+1). Инструкцией (int)L вычисляется код символа L, а затем, после увеличения кода символа на единицу, выполняется явное преобразование в символьный вид (инструкцией (char)). Для первого из двух создаваемых объектов номер объекта вычисляется на основе номера текущего объекта n как 2*n-1. Второй объект получает номер 2*n. Принцип нумерации рассчитан так, что если в вершине иерархии объект имеет номер 1, то на всех прочих уровнях объекты нумеруются последовательностью натуральных чисел. Таким образом, код конструктора класса реализован по рекурсивному принципу: в конструкторе вызывается конструктор, но с другими аргументами. Чтобы создать бинарное дерево, вызывается конструктор, первым аргументом которому передается количество уровней в бинарном дереве, имя (буква) для первого объекта и номер объекта в вершине иерархии объектов дерева.

В главном методе программы командой ObjectTree tree=new ObjectTree(4,'A',1) создается дерево из четырех уровней, после чего выполняется проверка созданной структуры: через систему последовательных ссылок вызывается метод show() для отображения параметров одного из объектов в структуре дерева (командой tree.FirstRef.SecondRef.FirstRef.show()). Результат выполнения программы может иметь следующий вид:

```
Создан новый объект!
Уровень объекта: A
Номер на уровне: 1
ID-код объекта: |3|5|1|1|1|1|5|0|
Создан новый объект!
Уровень объекта: B
Номер на уровне: 1
ID-код объекта: |3|6|4|8|2|2|2|9|
Создан новый объект!
Уровень объекта: C
Номер на уровне: 1
```

```
ID-код объекта: |7|6|7|8|9|1|5|7|
Создан новый объект!
Уровень объекта: D
Номер на уровне: 1
ID-код объекта: |5|6|6|1|6|6|5|4|
Создан новый объект!
Уровень объекта: D
Номер на уровне: 2
ID-код объекта: |3|6|5|0|2|1|6|7|
Создан новый объект!
Уровень объекта: C
Номер на уровне: 2
ID-код объекта: |7|6|0|9|6|1|0|2|
Создан новый объект!
Уровень объекта: D
Номер на уровне: 3
ID-код объекта: |9|2|6|2|5|5|9|9|
Создан новый объект!
Уровень объекта: D
Номер на уровне: 4
ID-код объекта: |8|7|7|0|1|2|1|4|
Создан новый объект!
Уровень объекта: B
Номер на уровне: 2
ID-код объекта: |1|7|3|8|8|1|9|2|
Создан новый объект!
Уровень объекта: C
Номер на уровне: 3
ID-код объекта: |9|3|2|4|7|9|4|7|
Создан новый объект!
Уровень объекта: D
Номер на уровне: 5
ID-код объекта: |7|9|6|4|9|4|4|4|
Создан новый объект!
Уровень объекта: D
Номер на уровне: 6
ID-код объекта: |0|9|5|4|5|5|7|4|
Создан новый объект!
Уровень объекта: C
Номер на уровне: 4
ID-код объекта: |4|1|6|6|9|7|8|1|
Создан новый объект!
Уровень объекта: D
Номер на уровне: 7
ID-код объекта: |6|4|7|5|0|0|0|3|
Создан новый объект!
Уровень объекта: D
Номер на уровне: 8
ID-код объекта: |3|9|9|6|7|3|6|3|
Проверка структуры дерева объектов!
Уровень объекта: D
Номер на уровне: 3
ID-код объекта: |9|2|6|2|5|5|9|9|
```

Сначала сверху вниз (см. рис.) создаются объекты, имеющие номер 1. Затем создается объект последнего уровня с номером 2. Далее создается объект предпоследнего уровня с номером 2, после чего объект предпоследнего уровня с номером 3, объект с номером 4 и т. д. Командой tree.FirstRef.SecondRef.FirstRef.show() метод show() вызывается из объекта уровня D с номером 3. Параметры именно этого объекта отображаются в конце программы.