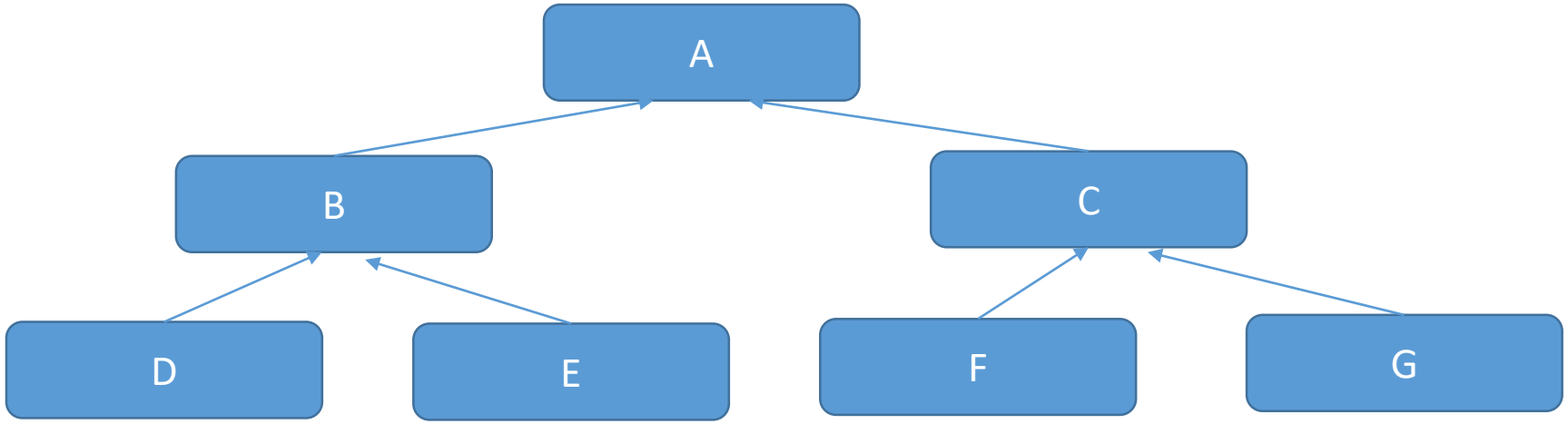
# Приведение ссылочных типов

При приведении ссылочных типов действуют следующие правила:

- объект всегда может быть приведен к типу своего непосредственного суперкласса;
- приведение ссылочных типов может выполняться по иерархии наследования классов сколь угодно глубоко;
- любой класс ссылочного типа всегда можно привести к типу Object.
- операция приведения типа имеет высокий приоритет.

1. После приведения объекта к типу суперкласса все переменные и методы самого класса объекта становятся недоступными для приведенного объекта.
2. Значение простого типа не может быть присвоено значению ссылочного типа, как и значение ссылочного типа не может быть присвоено значению простого типа.

# Расширяющее и сужающее приведение




**Расширяющее приведение** (или неявное)
A a1 = new F();
A a2 = new G();

**Сужающее приведение**(или явное)
F f1 =(F)a1;
G g2 = (G) a2;

A a3 = new F();
G g3 = (G) a3; // **Ошибочное приведение**
A a4 = new A ();
F f2 =(F) a4; // **Ошибочное приведение**

# Оператор instanceof

**С помощью оператора instanceof можно во время выполнения определить:**

- ✓К какому  типу принадлежит объект
- ✓Является ли объект подклассом указанного класса
- ✓Реализует ли объект интерфейс

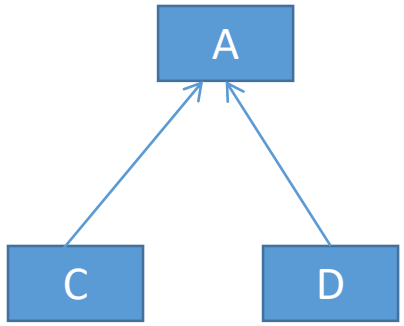
*Имя объекта* **instanceof** *Имя типа (класса)*

Левый операнд - любое выражение объектного типа, правый операнд - имя класса, интерфейса, массивный тип или String.

Если объект относится к указанному типу или создан на основе указанного класса (при создании объекта вызывался конструктор класса), то **оператор instanceof** дает true, а иначе - false.

# Пример использования оператора instanceof

```
class A {  int i, j; }
class C extends A {   int k; }
class D extends A {   int k; }
class InstanceOf {
  public static void main(String args[]) {
    A a = new A(); C c = new C();     D d = new D();
    if (a instanceof A)       System.out.println("a экземпляр A");
if (c instanceof A)       System.out.println("c создан на основе  A");
    if (a instanceof C)       System.out.println("a создан на основе C"); else  System.out.println(“a не создан на основе C");
        A ob;
    ob = d; // ob ссылается  на A часть объекта d
        if (ob instanceof D)       System.out.println("ob создан на основе D");
      ob = c; // ob ссылается  на A часть объекта c
        if (ob instanceof D)       System.out.println("ob создан на основе D");
    else       System.out.println("ob не создан на основе D");
    if (ob instanceof A)       System.out.println("ob создан на основе A");
        // все объекты созданы на основе Object
    if (a instanceof Object)       System.out.println(" a создан на основе Object");
} }
```

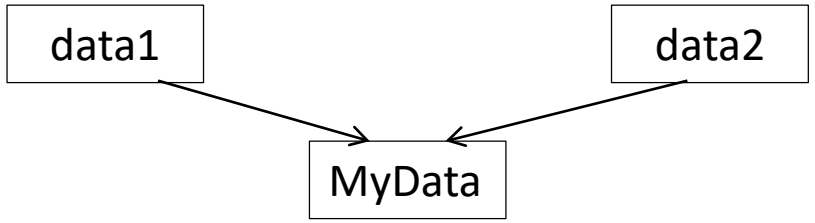


# Дублирование ссылок и Клонирование объектов

При обыкновенном присваивание объектов передаются ссылки на объект. В итоге два экземпляра ссылаются на один объект, и изменение одного приведет к изменению другого.

MyData data1 = new(MyData);  MyData data2 = data1;

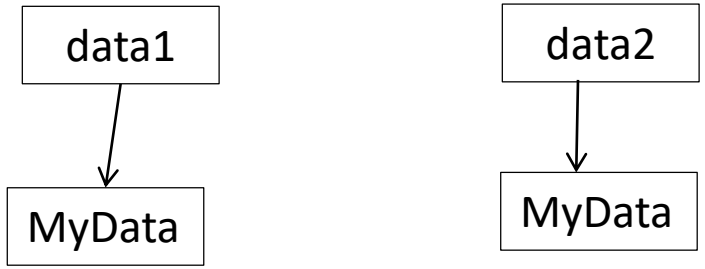
data2 = data1




Клонирование – порождение нового объекта независимого от  существующего.

MyData data2 = data1.clone();

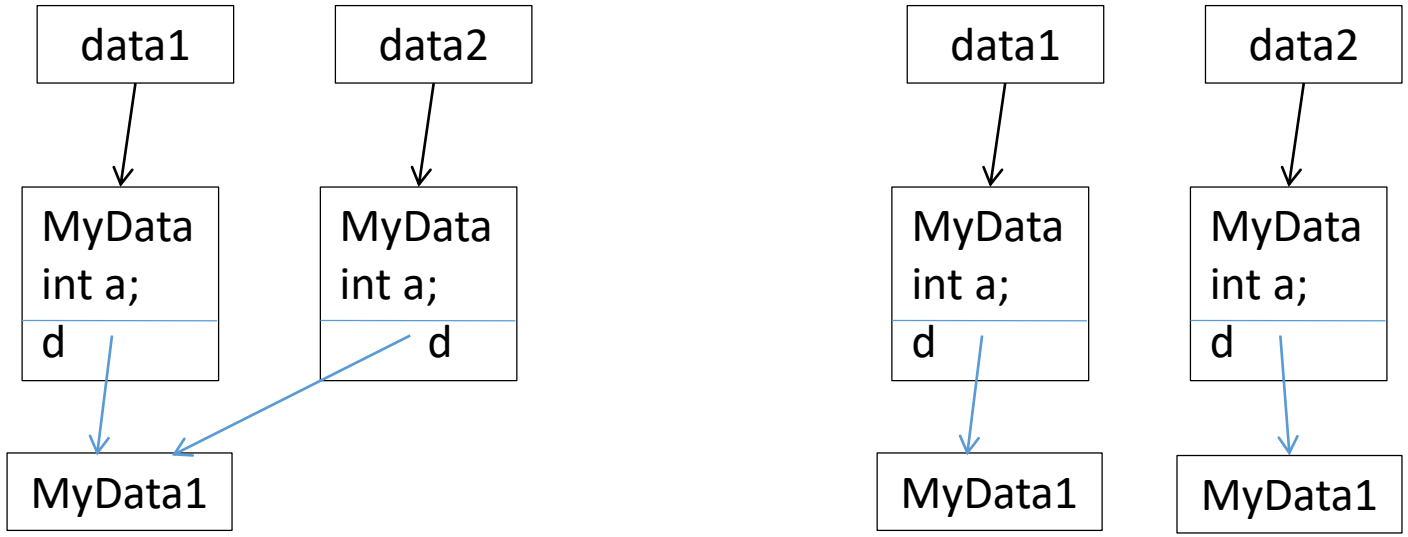
data2 ≠ data1



# Типы клонирования

Клонирование объекта бывает 2-х видов:

- Поверхностное (shallow); в этом случае копируются значения простых полей и ссылочные значения.
- Глубокое (deep); клонирование составного объекта, это достигается через рекурсивное клонирование составляющих его объектов.



# Способы клонирования

1. Переопределение метода clone() и реализация интерфейса Cloneable();
2. Использование конструктора копирования;
3. Использовать для клонирования механизм сериализации (заключается в сохранении объекта в поток байтов с последующим извлечением его от туда)

# Переопределение метода clone

В *Java* у базового класса *java.lang.Object* существует метод **clone**, с помощью которого можно создать новый объект точно такой же как и текущий.

Метод *clone* в классе *Object* является защищенным (*protected*)

**protected native Object clone() throws CloneNotSupportedException;**

**native**, говорит о том, что его реализация предоставляется виртуальной машиной.

Чтобы этот метод сделать общедоступным следует в описании подкласса указать интерфейс Cloneable и переопределить его с областью видимости *public*.

Интерфейс Cloneable не реализует ни одного метода. Он является всего лишь маркером, говорящим, что данный класс реализует клонирование объекта. Если класс не реализует данный интерфейс, то будет выброшено исключение *CloneNotSupportedException*.

```
public class MyData implements Cloneable {
        public int a;
     MyData1 d;
public MyData clone() throws CloneNotSupportedException {
        MyData obj = (MyData)super.clone();
// поверхностное копирование super.clone()  реализовано в классе Object, автоматически копируют примитивные значения
        return obj;      }    }
```

# Глубокое клонирование

```
public class MyData implements Cloneable {
        public int a;
    MyData1 d;
// переопределение метода clone()
// Метод clone() должен сначала вызвать super.clone, а затем исправить все ссылочные поля.
public MyData clone() throws CloneNotSupportedException {
   MyData obj = (MyData)super.clone(); // поверхностное копирование
    if (this.d != null)
   obj.d = this.d.clone(); // глубокое копирование
        return obj;
    }
  }
```

При клонировании в глубину нельзя использовать конструктор:

**obj.d= new MyData1();**

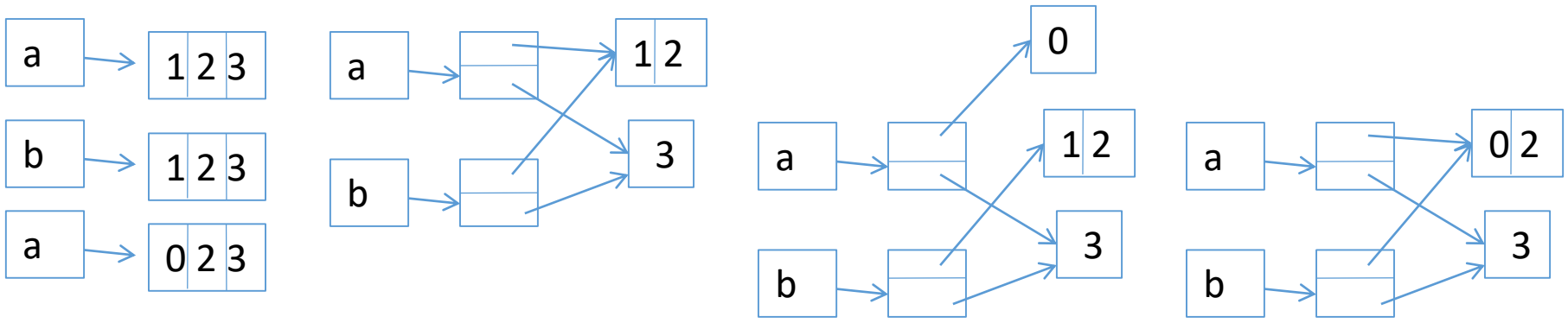
Поле d может содержать ссылку на объект класса **MyData1**

# Клонирование массивов

```
int a[]={1, 2, 3}; // один объект
int b[]=(int[])a.clone(); // поверхностное копирование
a[0]=0;
System.out.println(b[0]); // 1

int a[][]={{1, 2}, {3}}; // три объекта
int b[][]=(int[][]) a.clone();
{ a[0]=new int[]{0}; System.out.println(b[0][0]); }  // первый вариант: 1
{ a[0][0]=0; System.out.println(b[0][0]); }; // второй вариант: 0
}
```

***b[i] = a[i].clone();***



# Клонирование на основе конструктора копирования

```
public class Person {
private int age;
private String name;
public Person(int age, String name){
this.age=age; this.name=name; }
// конструктор копии с поверхностным клонированием
public Person(Person other) {
this(other.getAge(), other.getName());
}
// конструктор копии с глубоким клонированием
public Person(Person other) throws CloneNotSupportedException  {
this(other.getAge(), other.getName().clone()); //клонирование name
}
public int getAge() { return age; }
public String getName() { return name; }
}
```

# Понятие эквивалентности

Метод **equals()** обозначает отношение эквивалентности объектов. Эквивалентным называется отношение, которое является симметричным, транзитивным, рефлексивным и постоянным.

**Рефлексивность:** для любого ненулевого x, x.equals(x) вернет true;

**Транзитивность:** для любого ненулевого x, y и z, если x.equals(y) и y.eqals(z) вернет true, тогда и x.equals(z) вернет true;

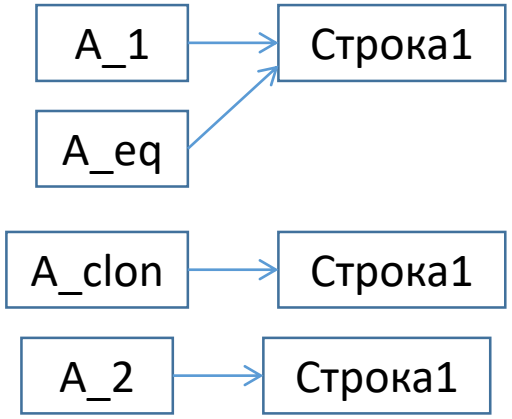
**Постоянство:** для любых объектов *x* и *y* x.equals(y) возвращает одно и тоже, если информация, используемая в сравнениях, не меняется;

**Симметричность:** для любого ненулевого x и y, x.equals(y) должно вернуть true, тогда и только тогда, когда y.equals(x) вернет true**.**

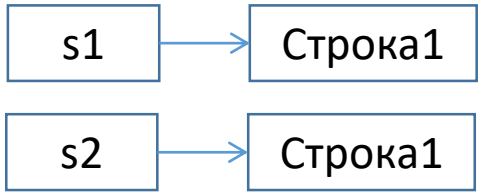
# Реализация equals в классе Object

Для определения равенства различных объектов применяется метод **equals**.
Метод **equals** реализован в классе Object и соответственно наследуем любым классом Java.
**public boolean equals(Object obj) { return (this == obj); }**
При сравнение объектов, операция “==” вернет true лишь в одном случае — когда ссылки указывают на один и тот же объект. В данном случае не учитывается содержимое полей (исключение String).

```
class A {
String objectName;
A (String name) { objectName = name; }      // Конструктор
}
public  class MyA {
public  static  void main (String args[ ])
{String s1 = new String(“Строка1 ");
String s2 = new String(“Строка1 "); String s3 = s1;
A  A_1 = new A("Строка1");      // Создание экземпляра класса
A  A_eq = A_1;       // Ссылка на существующий объект
A  A_clon = (A)A_1.clone;      // Создание объекта методом clone
A  A_2 = new A("Строка1");
 // Объекты-сравниваются ссылки
if (A_1.equals(A_eq))    { истина }
if (A_1.equals(A_ clon)) { ложь }
if (A_1.equals(A_ 2))    {  ложь}
} }
```




```
// Строки-сравниваются значения
if (s1.equals(s2)) { истина }
if (s1.equals(s3)) { истина }
if (s1==s2) { ложь }
```



# Переопределение метода equals

```
public class App
{  String str1 = new String("Test1");
String str2 = new String("Test2");
    int num = 5;
      public boolean equals(Object obj) { // переопределение equals
          if(obj == null) {return false;} // проверяет не равен ли obj – null
   if(!(obj instanceof App)){return false;} // проверяет является ли obj объектом App
    // или getClass() != obj.getClass()
                App obj1 = (App) obj;
          // сравнивает поля экземпляров класса
          return str1.equals(obj1.str1) && str2.equals(obj1.str2) && num == obj1.num;
      }
         public static void main( String[] args )
   {            App app1 = new App();
                App app2 = new App();
          System.out.println( app1.equals(app2) ); // результат:  true
System.out.println("хеш-код app1"+app1.hashCode()+ "хеш-код app2"+app2.hashCode());
 // хеш-код app1 10223330       хеш-код app2 24817388
```

# Хеш-код

Хеш-код – это номер для данного объекта в диапазоне от

-2 147 483 648 до 2 147 483 647. Данный номер позволяет быстро определить его местонахождение в коллекции и извлечь, а также сравнивать объекты.

По умолчанию, функция *hashCode()* для объекта возвращает номер ячейки памяти, где объект сохраняется.

1. Для одного и того же объекта, хеш-код всегда будет одинаковым.
2. Если объекты одинаковые, то и хеш-коды должны быть одинаковые (но не наоборот). Одинаковые объекты — это объекты одного класса с одинаковым содержимым полей.
3. Если хеш-коды равны, то входные объекты не всегда равны (коллизия).
4. Если хеш-коды разные, то и объекты гарантированно разные.
5. Если в классе переопределен метод equals(), то в этом классе надо переопределить метод hashCode().
6. Эквивалентность и хеш-код тесно связанны между собой, поскольку хеш-код вычисляется на основании содержимого объекта (значения полей) и если у двух объектов одного и того же класса содержимое одинаковое, то и хеш-коды должны быть одинаковые (см. правило 2).

# Алгоритм вычисления хеш-функции

1. Присвойте результирующей переменной (result) некоторое ненулевое простое число (например, 17)
2. Если поле value имеет тип boolean, вычислите (value ? O : 1)
3. Если поле value имеет тип byte, char, short или int, вычислите (int)value
4. Если поле value имеет тип long, вычислите (int)(value - (value >>> 32))
5. Если поле value имеет тип float, вычислите Float.floatToIntBits(value)
6. Если поле value имеет тип double, вычислите Double.doubleToLongBits(value), а затем преобразуйте полученное значение, как указано в п.4
7. Если поле value является ссылкой на объект, вызывайте метод hashCode() этого объекта (для string метод переопределен)
8. Если поле value является ссылкой на объект и равно null, используйте число 0 для представления его хэш-кода
9. Объедините полученные в п. 2 - п. 8 значения следующим образом: 37*result + value
10. Если поле является массивом, примените правило 9 для каждого элемента массива
11. Проверьте, что равные объекты возвращают одинаковый hashCode

# Хеш-функция для класса App

```
public int hashCode() {
        int result = 17;
        result = 37 * result + (int) num;
result = 37 * result + (str1 == null ? 0 : str1.hashCode());
result = 37 * result + (str2 == null ? 0 : str2.hashCode());
        return result;
    }
App app1 = new App(); app1.hashCode();
App app2 = new App(); app2.hashCode()
```

хеш-код app1-1227544379

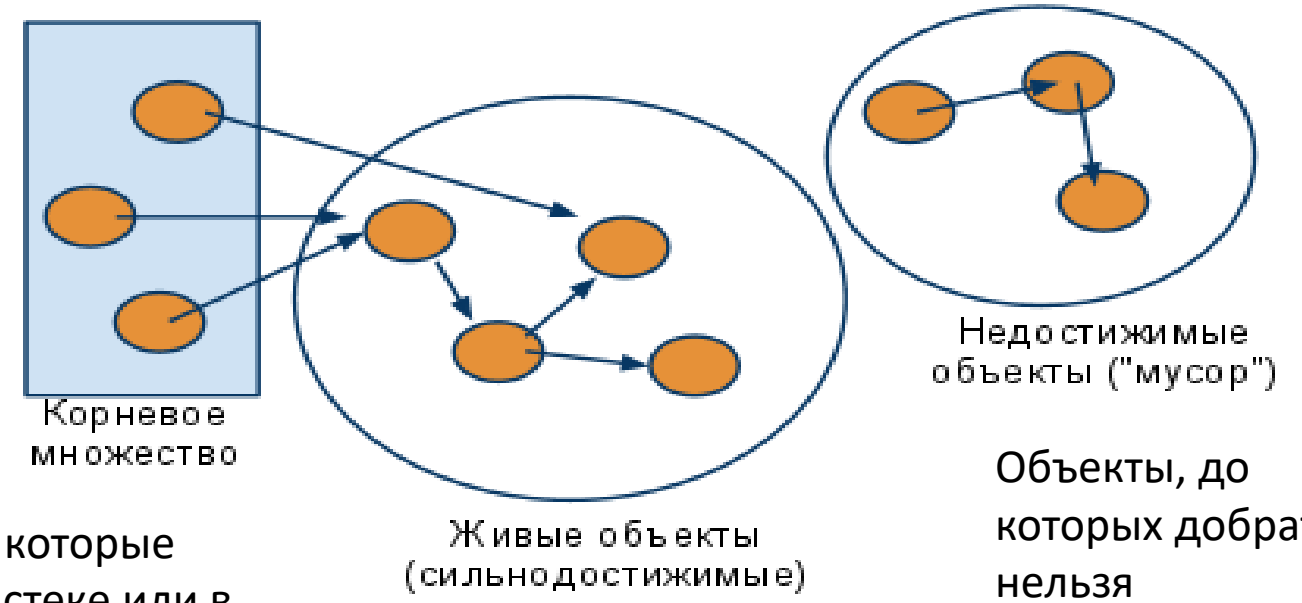
хеш-код app2-1227544379

```
public class App
{  String str1 = new tring("Test1");
String str2 = new String("Test2");
int num = 5;
}
```

Вычисление хеш-кода строки

s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1], где s[i] — i-ый символ строки, n — длина строки, и ^ указывает на возведение в степень. (Хэш-значение пустой строки равно нулю).

# Управление памятью




Все ссылки, которые находятся в стеке или в статической области памяти

Все объекты в «куче», до которых можно добраться по ссылкам из корневого множества

Объекты, до которых добраться нельзя

# Виды ссылок в Java

- **Strong** (жесткая, сильная)Любой объект, который имеет strong ссылку запрещен для удаления сборщиком мусора.

Rectangle rect = new Rectangle();

- **Weak** (слабая) Объект будет удален, если на него не ссылаются сильные и мягкие ссылки.

WeakReference<Rectangle> rect =

new WeakReference<Rectangle>(new Rectangle());

- **Soft** (мягкая) Объект будет удален в случае, если JVM требуется память.

SoftReference<Rectangle> rect =

new SoftReference<Rectangle>(new Rectangle());

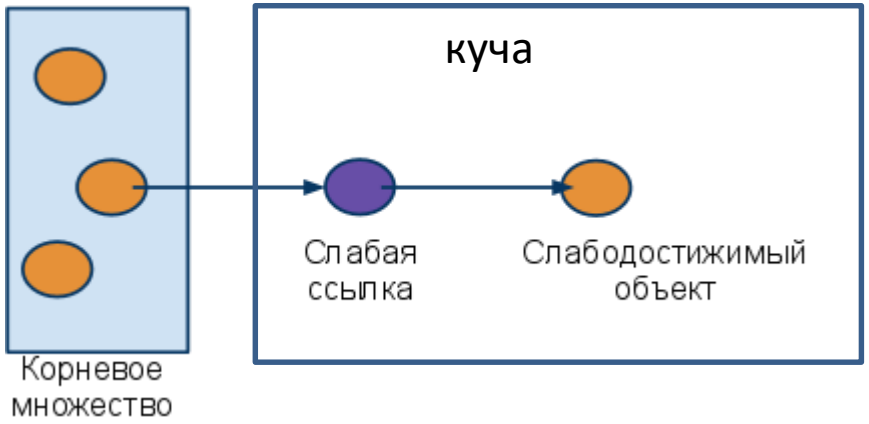
- **Phantom** (призрачная) Объект на который указывают только phantom ссылки может быть удален сборщиком в любой момент, при этом можно получить уведомление о помещении ссылки в очередь на удаление.

PhantomReference<Rectangle> rect =

new PhantomReference<Rectangle>(new Rectangle(), queue);

# Слабодостижимые объекты

Объект называется слабодостижимым, если до него можно добраться только по слабым ссылкам из корневого множества.



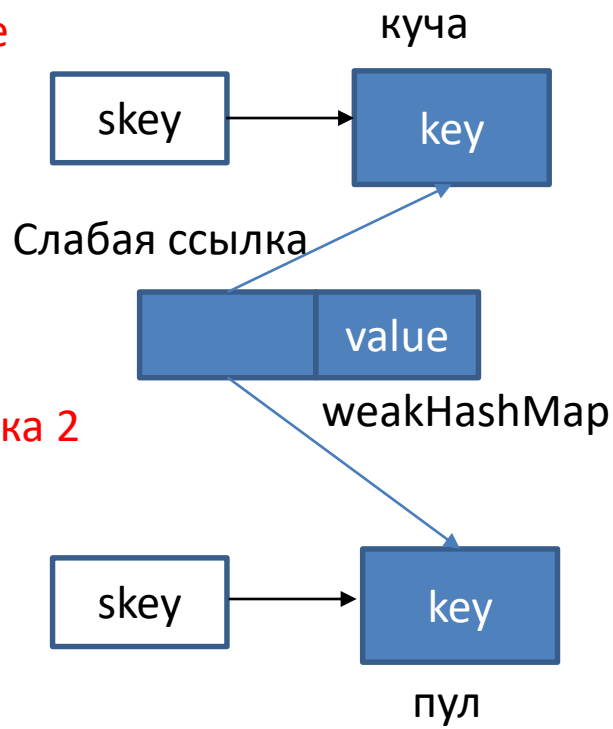

Слабые ссылки отличаются от обычных тем, что не препятствуют удалению объекта из памяти, т.е. если остаются только слабые ссылки, то объект будет уничтожен при ближайшей сборке мусора

WeakHashMap - это структура данных, реализующая интерфейс Map и основанная на использовании  слабых ссылок для хранения ключей. Пара "ключ-значение" будет удалена из WeakHashMap, если на объект-ключ не имеется сильных ссылок.

# Пример использования слабых ссылок

```
import java.util.*;
public class WeakHashMapDemo {
public static void main(String[] args) {
// создаем объект со слабыми ссылками на ключи
Map<String, String> weakHashMap = new WeakHashMap<String, String>();
// создаем строку с жесткой ссылкой
String skey = new String("key");
//String skey = "key"; // потом попробуйте так и сравните результат
// помещаем какое-то значение
weakHashMap.put(skey, "value");
// вызываем  мусорщик
System.gc();
// выводим результат, наше значение на месте
System.out.println(weakHashMap.get("key"));
// удаляем жесткую ссылку, теперь "key" стал слабо-доступным
skey = null;
// вызываем  мусорщик еще раз
System.gc();  // закомментируйте и сравните результат
// элемент исчезнет из отображения при сборке мусора
System.out.println(weakHashMap.get("key"));
} }
```

Строка в куче
Без вызова мусорщика 2
value
value




Строка в куче
Вызов мусорщика 2
value
null

Строка в пуле
Вызов мусорщика 2
value
value