В то же время такая реализация метода **equals()** будет возвращать истину при сравнении объектов класса **SubStudent** с одинаковыми значениями полей, унаследованных от класса **Student**.

Клонирование объектов

Объекты в методы передаются по ссылке, в результате чего в метод передается ссылка на объект, находящийся вне метода. Поэтому если в методе изменить значение поля объекта, то это изменение коснется исходного объекта. Во избежание такой ситуации для защиты внешнего объекта следует создать клон (копию) объекта в методе. Класс **Object** содержит **protected**-метод **clone()**, осуществляющий побитовое копирование объекта производного класса. Однако сначала необходимо переопределить метод **clone()** как **public** для обеспечения возможности вызова из другого пакета. В переопределенном методе следует вызвать базовую версию метода **super.clone()**, которая и выполняет собственно клонирование. Чтобы окончательно сделать объект клонируемым, класс должен реализовать интерфейс **Cloneable**. Интерфейс **Cloneable** не содержит методов относится к помеченным (tagged) интерфейсам, а его реализация гарантирует, что метод **clone()** класса **Object** возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение **CloneNotSupportedException**. Следует отметить, что при использовании этого механизма объект создается без вызова конструктора. В языке C++ аналогичный механизм реализован с помощью конструктора копирования.

```
/* пример # 13 : класс, поддерживающий клонирование: Student.java */
package chapt04;

public class Student implements Cloneable {/*включение
                                                интерфейса */
      private int id = 71;

      public int getId() {
            return id;
      }
      public void setId(int value) {
            id = value;
      }
      public Object clone() {//переопределение метода
            try {
                  return super.clone();//вызов базового метода
            } catch (CloneNotSupportedException e) {
                  throw new AssertionError("невозможно!");
            }
      }
}
```

```
/* пример # 14 : безопасная передача по ссылке: DemoSimpleClone.java */
package chapt04;
```

```
public class DemoSimpleClone {
      private static void changeId(Student p) {
            p = (Student) p.clone();//клонирование
            p.setId(1000);
            System.out.println("->id = " + p.getId());
            }
      public static void main(String[] args) {
            Student ob = new Student();
            System.out.println("id = " + ob.getId());
            changeId(ob);
            System.out.println("id = " + ob.getId());
      }
}
```

В результате будет выведено:

```
id = 71
->id = 1000
id = 71
```

Если закомментировать вызов метода **clone()**, то выведено будет следующее:

```
id = 71
->id = 1000
id = 1000
```

Такое решение эффективно только в случае, если поля клонируемого объекта представляют собой значения базовых типов и их оболочек или неизменяемых (immutable) объектных типов. Если же поле клонируемого типа является изменяемым объектным типом, то для корректного клонирования требуется другой подход. Причина заключается в том, что при создании копии поля оригинал и копия представляют собой ссылку на один и тот же объект. В этой ситуации следует также клонировать и объект поля класса.

```
/* пример # 15 : глубокое клонирование: Student.java */
package chapt04;
import java.util.ArrayList;

public class Student implements Cloneable {
      private int id = 71;
      private ArrayList<Mark> lm = new ArrayList<Mark>();

      public int getId() {
            return id;
      }
      public void setId(int id) {
            this.id = id;
      }
      public ArrayList<Mark> getMark() {
            return lm;
      }
```

```
    public void setMark(ArrayList<Mark> lm) {
         this.lm = lm;
    }
    public Object clone() {
          try {
                Student copy =(Student)super.clone();
                copy.lm = (ArrayList<Mark>)lm.clone();
                return copy;
          } catch (CloneNotSupportedException e) {
                throw new AssertionError(
                            "отсутствует Cloneable!");
          }
    }
}
```

Такое клонирование возможно только в случае, если тип атрибута класса также реализует интерфейс **Cloneable** и переопределяет метод **clone()**. В противном случае вызов метода невозможен, так как он просто недоступен. Следовательно, если класс имеет суперкласс, то для реализации механизма клонирования текущего класса необходимо наличие корректной реализации такого механизма в суперклассе. При этом следует отказаться от использования объявлений **final** для полей объектных типов по причине невозможности изменения их значений при реализации клонирования.

"Сборка мусора" и освобождение ресурсов

Так как объекты создаются динамически с помощью операции **new**, а уничтожаются автоматически, то желательно знать механизм ликвидации объектов и способ освобождения памяти. Автоматическое освобождение памяти, занимаемой объектом, выполняется с помощью механизма "сборки мусора". Когда никаких ссылок на объект не существует, то есть все ссылки на него вышли из области видимости программы, предполагается, что объект больше не нужен, и память, занятая объектом, может быть освобождена. "Сборка мусора" происходит нерегулярно во время выполнения программы. Форсировать "сборку мусора" невозможно, можно лишь "рекомендовать" ее выполнить вызовом метода **System.gc()** или **Runtime.getRuntime().gc()**, но виртуальная машина выполнит очистку памяти тогда, когда сама посчитает это удобным. Вызов метода **System.runFinalization()** приведет к запуску метода **finalize()** для объектов утративших все ссылки.

Иногда объекту нужно выполнять некоторые действия перед освобождением памяти. Например, освободить внешние ресурсы. Для обработки таких ситуаций могут применяться два способа: конструкция **try-finally** и механизм finalization. Конструкция **try-finally** является предпочтительной, абсолютно надежной и будет рассмотрена в девятой главе. Запуск механизма finalization определяется алгоритмом сборки мусора и до его непосредственного исполнения может пройти сколь угодно много времени. Из-за всего этого поведение метода **finalize()** может повлиять на корректную работу программы, особенно при смене JVM. Если существует возможность освободить ресурсы или выполнить другие подобные действия без привлечения этого механизма, то лучше без него