В процессе выполнения приложения будет случайным образом сформирован массив-тест из вопросов разного типа, будет выполнена проверка теста, а общая информация об ответах на них будет выведена на консоль:

27 correct answers, 33 incorrect

Класс **QuestFactory** содержит метод **getQuestFromFactory(int numMode)**, который возвращает ссылку на случайно выбранный объект подкласса класса **AbstractQuest** каждый раз, когда он вызывается. Приведение к базовому типу производится оператором **return**, который возвращает ссылку на **DragnDropQuest** или **SingleChoiceQuest**. Метод **main()** содержит массив из ссылок **AbstractQuest**, заполненный с помощью вызова **getQuestFromFactory()**. На этом этапе известно, что имеется некоторое множество ссылок на объекты базового типа и ничего больше (не больше, чем знает компилятор). Когда происходит перемещение по этому массиву, метод **check()** вызывается для каждого случайным образом выбранного объекта.

Если понадобится в дальнейшем добавить в систему, например, класс **MultiplyChoiceQuest**, то это потребует только переопределения метода **check()** и добавления одной строки в код метода **getQuestFromFactory()**, что делает систему легко расширяемой.

Невозможно приравнивать ссылки на классы, находящиеся в разных ветвях наследования, так как не существует никакого способа привести один такой тип к другому.

Класс Object

На вершине иерархии классов находится класс **Object**, суперкласс для всех классов. Изучение класса **Object** и его методов необходимо, т.к. его свойствами обладают все классы Java. Ссылочная переменная типа **Object** может указывать на объект любого другого класса, на любой массив, так как массив реализован как класс-наследник **Object**.

В классе **Object** определен набор методов, который наследуется всеми классами:

- **protected Object clone()** — создает и возвращает копию вызывающего объекта;
- **public boolean equals(Object ob)** — предназначен для использования и переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов одного и того же типа;
- **public Class<? extends Object> getClass()** — возвращает экземпляр типа **Class**;
- **protected void finalize()** — (deprecated) автоматически вызывается сборщиком мусора (garbage collection) перед уничтожением объекта;
- **public int hashCode()** — вычисляет и возвращает хэш-код объекта (число, в общем случае вычисляемое на основе значений полей объекта);
- **public String toString()** — возвращает представление объекта в виде строки.

Методы **notify()**, **notifyAll()** и **wait()**, **wait(int millis)** будут рассмотрены в главе «Потоки выполнения».

Если при создании класса предполагается проверка логической эквивалентности объектов, которая не выполнена в суперклассе, следует переопределить два метода: **boolean equals(Object ob)** и **int hashCode()**. Кроме того, переопределение этих методов необходимо, если логика приложения предусматривает использование элементов в коллекциях. Метод **equals()** при сравнении двух объектов возвращает истину, если содержимое объектов эквивалентно, и ложь — в противном случае. Реализация метода в классе **Object** возвращает истину только в том случае, если обе ссылки указывают на один и тот же объект, а конкретно:

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

При переопределении метода **equals()** должны выполняться соглашения, предусмотренные спецификацией языка Java, а именно:

- рефлексивность — объект равен самому себе;
- симметричность — если **x.equals(y)** возвращает значение **true**, то и **y.equals(x)** всегда возвращает значение **true**;
- транзитивность — если метод **equals()** возвращает значение **true** при сравнении объектов **x** и **y**, а также **y** и **z**, то и при сравнении **x** и **z** будет возвращено значение **true**;
- непротиворечивость — при многократном вызове метода для двух не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;
- ненулевая ссылка при сравнении с литералом **null** всегда возвращает значение **false**.

При создании информационных классов также рекомендуется переопределять методы **hashCode()** и **toString()**, чтобы адаптировать их действия для создаваемого типа.

Метод **int hashCode()** переопределен, как правило, в каждом классе и возвращает число, являющееся уникальным идентификатором объекта, зависящим в большинстве случаев только от значения объекта. Его следует переопределять всегда, когда переопределен метод **equals()**. Метод **hashCode()** возвращает хэш-код объекта, вычисление которого управляется следующими соглашениями:

- все одинаковые по содержанию объекты *одного типа* **должны** иметь одинаковые хэш-коды;
- различные по содержанию объекты *одного типа* **могут** иметь различные хэш-коды;
- во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен.

Один из способов создания правильного метода **hashCode()**, гарантирующий выполнение соглашений, приведен ниже для класса **Student**.

Метод **toString()** следует переопределять таким образом, чтобы, кроме стандартной информации о пакете (опционально), в котором находится класс, и самого имени класса (опционально), он возвращал значения полей объекта, вызвавшего этот метод (т.е. всю полезную информацию объекта), вместо хэш-кода, как это делается в классе **Object**. Метод **toString()** класса **Object** возвращает строку с описанием объекта в виде:

getClass().getName() + '@' + Integer.toHexString(hashCode())

Метод вызывается автоматически, когда объект передается в поток вывода методами **println()**, **print()** и некоторыми другими.

/ # 17 # переопределение методов equals(), hashCode(), toString() # Student.java */*

```
package by.epam.learn.entity;
public class Student {
  private int id;
  private String name;
  private int yearOfStudy;
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public int getYearOfStudy() {
    return yearOfStudy;
  }
  public void setYearOfStudy(int yearOfStudy) {
    this.yearOfStudy = yearOfStudy;
  }
  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (o == null || getClass() != o.getClass()) {
      return false;
    }
    Student student = (Student) o;
    if (id != student.id) {
```

```
      return false;
    }
    if (yearOfStudy != student.yearOfStudy) {
      return false;
    }
    return name != null ? name.equals(student.name) : student.name == null;
  }
  @Override
  public int hashCode() {
    return id + 31 * yearOfStudy + (name != null ? name.hashCode() : 0) ;
  }

  @Override
  public String toString() {
    final StringBuilder sb = new StringBuilder("Student{");
    sb.append("id=").append(id);
    sb.append(", name='").append(name).append('\'');
    sb.append(", yearOfStudy=").append(yearOfStudy);
    sb.append('}');
    return sb.toString();
  }
}
```

Выражение **id + 31 * yearOfStudy** гарантирует различные результаты вычислений при перемене местами значений полей, а именно: если **id=1** и **yearOfStudy=2**, то в результате будет получено **33**, если значения поменять местами, то **63**. Такой подход применяется при наличии у классов полей базовых типов.

Формально всем трем правилам определения метода **hashCode()** удовлетворяет и такая реализация:

```
public int hashCode() {
  return 42;
}
```

но все же следует признавать корректной ту реализацию метода, при которой для различных по содержанию объектов значения хэш-кодов были различными.

Метод **equals()** переопределяется для класса **Student** таким образом, чтобы убедиться в том, что полученный объект является объектом типа **Student**, а также сравнить содержимое полей **id**, **name** и **yearOfStudy** соответственно у вызывающего метод объекта и объекта, передаваемого в качестве параметра. Для подкласса всегда придется создавать собственную реализацию метода.

Клонирование объектов

Объекты в методы передаются по ссылке, в результате чего метод получает ссылку на объект, находящийся вне метода. Если в методе изменить значение поля объекта, 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**.

При использовании этого механизма объект создается без вызова конструктора, на уровне виртуальной машины выполняется побитовое копирование объекта в другую часть памяти. В языке C++ аналогичный механизм реализован с помощью конструктора копирования.

Чтобы продемонстрировать реализацию клонирования классу **Student** следует добавить имплементацию интерфейса **Cloneable**:

```
public class Student implements Cloneable
```

а в сам класс реализацию метода **clone()**:

```
    @Override
    public Object clone() throws CloneNotSupportedException {
      return super.clone();
    }
```

Тогда безопасное клонирование можно представить в виде:

/ # 18 # безопасная передача по ссылке # CloneMain.java */*

```
public class CloneMain {
  private static void preparation(Student student) {
    try {
      student = (Student) student.clone(); // cloning
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
    student.setId(1000);
    System.out.println("->id = " + student.getId());
  }
  public static void main(String[] args) {
    Student ob = new Student();
    ob.setId(71);
    System.out.println("id = " + ob.getId());
    preparation(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

То есть отключение клонирования привело к тому, что изменение состояния объекта в методе отразилось и на исходном объекте.

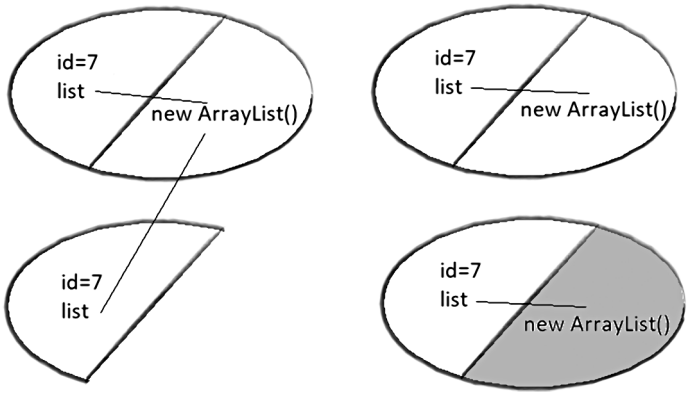

Рис. 4.1. *«Неглубокое» и «глубокое» клонирование*

Решение эффективно только в случае, когда 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, если он сам поддерживает клонирование.

/ # 19 # глубокое клонирование # Abiturient.java */*

```
package by.epam.learn.entity;
import java.util.ArrayList;
public class Abiturient implements Cloneable {
  private int id = 7;
  private ArrayList<Byte> list = new ArrayList<>();
```

```
  public ArrayList<Byte> getList() {
    return list;
  }
  public void setList(ArrayList<Byte> list) {
    this.list = list;
  }
  @Override
  public Abiturient clone() {
    Abiturient copy = null;
    try {
      copy = (Abiturient)super.clone();
      copy.list = (ArrayList<Byte>) list.clone();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
    return copy;
  }
  @Override
  public String toString() {
    final StringBuilder sb = new StringBuilder("Abiturient{");
    sb.append("id=").append(id);
    sb.append(", list=").append(list);
    sb.append('}');
    return sb.toString();
  }
}
```

Клонирование возможно лишь, если тип атрибута класса также реализует интерфейс **Cloneable** и переопределяет метод **clone()**. В противном случае вызов метода невозможен, так как он просто недоступен.

/ # 20 # демонстрация глубокого клонирования # CloneMain.java */*

```
package by.epam.learn.main;
import by.epam.learn.entity.Abiturient;
import java.util.ArrayList;
public class CloneMain {
  public static void main(String[] args) {
    Abiturient abiturient = new Abiturient();
    ArrayList<Byte> list = new ArrayList<>();
    list.add((byte)1);
    list.add((byte)2);
    abiturient.setList(list);
    Abiturient abiturient1 = abiturient.clone();
    list = abiturient1.getList();
    list.remove(0);
    list.add((byte)9);
    System.out.println(abiturient);
    System.out.println(abiturient1);
  }
}
```

Клонированный объект не имеет никаких зависимостей от оригинала:

```
Abiturient{id=7, list=[1, 2]}
Abiturient{id=7, list=[2, 9]}
```

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.

Если заменить объявление **ArrayList<Byte>** на **CustomCollection<Task>**, где **Task** — изменяемый тип, то клонирование должно затрагивать и внутреннее состояние коллекции.

«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, т.е. 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» выполнить ее 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 ссылки. Начиная с версии Java 9 метод **finalize()** помечен как deprecated, тем самым не рекомендован к использованию, как и не был рекомендован еще в первые годы существования языка Java, но механизм его работы важен для понимания принципов работы «сборщика мусора».

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 autocloseable. Указанные способы являются предпочтительными, абсолютно надежными и будут рассмотрены в девятой главе.

Запуск стандартного 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 обойтись.