Язык программирования Java

Лекция 2

Перевод курса CS11 Java Track

Содержание

- Пакеты
- Интерфейсы
- Классы коллекций

Пакеты Java

- Классы можно группировать в пакеты (packages)
  - Пакет это коллекция логически связанных типов
- Пакеты используют пространства имен
  - В одном пространстве имен не может быть двух классов с одинаковыми именами
  - Но классы могут иметь одинаковые имена, если они находятся в разных пакетах
- Если пакет класса не указан явно, класс помещается в “пакет по умолчанию”
  - Пакет по умолчанию не имеет имени
- Для указания имени пакета используется директива package:

```
package cs11;
```

  - Директива должна быть первой в .java файле
  - Она определяет место куда помещаются файлы .java и .class
  - Пока не используйте директиву package в своих программах

Работа с классами в пакетах

- Если класс находится в пакете, то к нему можно обращаться используя его полное имя

```
java.util.ArrayList myList = new java.util.ArrayList();
```

- Надо импортировать класс

```
import java.util.ArrayList;
...
ArrayList myList = new ArrayList();
```

- Или импортировать весь пакет

```
import java.util.*;
...
ArrayList myList = new ArrayList();
```

Пакеты Java API

- Все классы Java API размещены в пакетах.
- Классы пакета java.lang импортируются автоматически
- Для импорта классов из других пакетов следует использовать директиву import

```
import java.util.ArrayList;
import java.util.HashSet;
...
```

  - Или так

```
import java.util.*;
```

- Импорт пакетов не работает рекурсивно
  - Импорт java.* не добавит к программе ничего

Поведение объекта

- Часто необходимо иметь классы, по разному реализующие внешне одинаковое поведение
  - Напомним, что поведение объекта определяется набором его методов
  - Наш случай предполагает в классах одинаковый набор методов имеющих разную внутреннюю реализацию
- Интерфейсы похожи на классы, но содержат только сигнатуры методов
  - Интерфейсы содержат только декларацию поведения, но не содержат определение поведения
  - В интерфейсах отсутствует реализация методов, а также поля данных
- Класс может реализовывать несколько интерфейсов
  - В Java это называется множественное наследование интерфейсов
  - Заметим, что в Java не поддерживается множественное наследование классов

Интерфейсы

- Интерфейс “определяет протокол взаимодействия между объектами”
  - В интерфейсе объявляется набор методов (поведений)
- Класс, реализующий интерфейс должен содержать реализации всех методов интерфейса
- Работа с объектом происходит черерез его интерфейс
  - Детали и особенности реализации поведения объекта скрыты от его клиентов

- Интерфейсы объявляются также как классы

```
/** базовый компонент моделирования. */
public interface SimComponent {
    /** Инициализация компонента. */
    void init(SimConfig sconf);
    /** Выполнить моделирование. */
    void simulate(double timestep);
    /** Завершить моделирование. */
    void shutdown();
}
```

- Код помещается в SimComponent.java
- Модификаторы доступа к интерфейсам не применяются. Доступ всегда public.

Интерфейсы и классы

- Классы могут реализовывать интерфейсы
  - С экземпляром такого класса можно работать, как будто он имеет тип интерфейса, который этот класс реализует
  - Класс может реализовывать любое количество интерфейсов
  - Так реализуется более простой и понятный механизм множественного наследования
- Интерфейсы не поддерживают наследование
  - Нельзя наследовать интерфейс от интерфейса, но интерфейс может быть реализован классом, а этот класс может иметь класс наследник
- Тип переменной может быть интерфейсом, так же как и классом

Реализация интерфейсов

- Реализация метода интерфейса в классе должна обязательно иметь модификатор доступа public

```
public class PhysicsEngine implements SimComponent {
    ...
    public void init(SimConfig simConf) {
    ... // делаем здесь что то
    }
    public void simulate(double timestep) {
        ... // делаем здесь что то еще
    }
    ...
}
```

- Любой может обратиться к интерфейсу класса потому, что он public

Использование интерфейсов

- Интерфейсы следует использовать для разделения программы на компоненты
  - … особенно важно делать это, если какой либо компонент может иметь различные реализации!
  - Компоненты будут взаимодействовать друг с другом через общий интерфейс, не заботясь о внутренних деталях реализации этого интерфейса в разных компонентах.
- Пример, хранение календарных событий пользователя

```
public interface CalendarStorage {
    // Загрузить события пользователя
    Calendar loadCalendar(String username);
    // Сохранить календарь пользователя
    void saveCalendar(String username, Calendar c);
}
```

Использование интерфейсов (2)

- Сделаем разные реализации хранения событий календаря
  - в локальном файле данных:

```
public class FileCalendarStorage
    implements CalendarStorage {
    ...
}
```

  - и на удаленном сервере:

```
public class
RemoteCalendarStorage
    implements CalendarStorage {
    ...
}
```

- Теперь напишем код, работающий с интерфейсом, а не с его реализациями:

```
CalendarStorage calStore = openCalendar();
Calendar cal =
calStore.loadCalendar(username);
```

Использование интерфейсов (3)

- В такой схеме можно менять при необходимости детали реализации
  - …следя, однако, за тем чтобы интерфейс оставался неизменным
- Если интерфейс становится большим и сложным
  - Можно использовать “заглушки”, на этапе разработки, пока интерфейс реализован не полностью

```
public class FakeCalendarStorage
    implements CalendarStorage {
    public Calendar loadCalendar(String username) {
        return Calendar(username); // пустой календарь
    }
    public void saveCalendar(String username,Calendar c){
        // ничего не делаем!
    }
}
```

- Это позволяет параллельно разрабатывать зависимые компоненты программы

Расширение интерфейсов

- Интерфейсы можно расширять:

```
/** Компонент моделирования работающий по сети. */
public interface DistributedSimComponent
    extends SimComponent {
    /** Подключиться к серверу. */
    void connect(String hostname);
    /** Отключиться от сервера. */
    void disconnect();
}
```

- Интерфейс из этого примера включает все методы, объявленные в интерфейсе SimComponent
- Все методы, по прежнему, имеют тип доступа public

Коллекции Java

- Java имеет очень полезный набор классов для работы с коллекциями
  - Добавлены в версии Java 1.2
- Эти классы предлагают:
  - Интерфейсы для работы с различными типами коллекций
  - Различные реализации, обладающие разными характеристиками
  - Итераторы, предназначенные для перемещения по элементам коллекций
  - Реализации некоторых общих алгоритмов работы с коллекциями
- Следует отметить, что по производительности и гибкости эти классы, конечно же, уступают C++ STL

Зачем нужны классы коллекций?

- Классы для работы с коллекциями существенно облегчают программирование
  - Большинство программ, так или иначе, используют коллекции
  - Эти классы делают язык Java более удобным для разработчиков
- Стандартный набор интерфейсов и свойств классов коллекций
  - Облегчает изучение
  - Упрощает взаимодействие между API
- Java API обеспечивает высоко производительную эффективную и корректную реализацию методов работы с коллекциями для пользователей

Интерфейсы коллекций

- Базовые интерфейсы коллекций объявлены в пакете java.util
  - Здесь содержится основной функционал для всех типов коллекций
- Collection – базовый набор объектов
- List (список) – линейный список элементов, доступных по индексу
- Queue (очередь) – линейный список элементов “для обработки”
  - В очередь можно добавить элемент
  - Из очереди можно получить “следующий элемент”
  - Какой из элементов “следующий ” зависит от реализации очереди
- Set (множество) – коллекция не содержащая повторяющихся элементов
- Map (ассоциативный список) – список в котором значения ассоциируются с ключами

Другие интерфейсы коллекций

- Есть интерфейсы с дополнительными свойствами
  - SortedSet (расширение Set)
  - SortedMap (расширение Map)
    - Эти коллекции гарантированно располагают элементы в заданном порядке
- Элементы в таких коллекциях должны быть сравнимы
  - Это означает, что для пары элементов, всегда можно сказать какой из них “больше” или они “равны” друг другу
  - Для элементов такой коллекции должна существовать общая процедура сортировки (выстраивания по порядку)

Общие операции коллекций

- Коллекции, как правило, реализуют следующий набор операций:
  - add(Object o) – добавление элемента к коллекции
  - remove(Object o) – удаление элемента
  - clear() – удаление всех элементов коллекции
  - size() – возвращает число элементов в коллекции
  - isEmpty() - возвращает true, если коллекции пуста
  - iterator() – перебирает элементы коллекции
- Некоторые операции не обязательны для всех коллекций
  - Если коллекция не поддерживает операцию, вызывается исключение UnsupportedOperationException
- Операции отличаются по скорости исполнения

Реализации коллекций

- Есть несколько реализаций каждого интерфейса
  - Все коллекции реализуют базовый набор функций
  - Коллекции реализуют разные способы хранения данных
  - Коллекции имеют разную производительность
  - Некоторые коллекции имеют дополнительные функции
    - Которые могут не входить в интерфейс
- Детальное описание можно найти в документации Java API
  - В документации на интерфейс можно найти список его реализаций
  - В документации на каждую реализацию можно найти детали касающиеся производительности и способов хранения данных

Реализации списков

- LinkedList – двунаправленный связный список
  - Каждый элемент имеет ссылку на предыдущий и последующий элемент
  - Время доступа к элементу списка зависит от его позиции
  - Время вставки и добавления элементов не зависит от их позиции
  - Элементы хранят не только значение, но и служебную информацию (указатели на следующий и предыдущий элементы и др.)
  - Такие списки удобно использовать, если количество элементов изменяется во время работы
  - Класс имеет дополнительные функции для извлечения/удаления первого и последнего элементов.
- ArrayList – хранит элементы в массиве
  - Время доступа к элементу не зависит от его позиции в массиве
  - Время добавления обычно не зависит от позиции
  - Рекомендуется использовать в случае, если размер массива не часто меняется во время работы
  - Класс имеет дополнительные методы для преобразования в обычный массив

Реализация множеств

- HashSet
  - Элементы группируются в “корзины” по хэш коду
  - Время добавления/удаления элементов не зависит от позиции
  - Время проверки присутствия элемента в списке не зависит от позиции элемента
  - Элементы не хранятся, в каком либо порядке
  - Для каждого элемента должна существовать функция вычисления хэша.
- TreeSet
  - Отсортированные элементы хранятся в сбалансированном дереве
  - Время добавления/удаления и поиска элементов возрастает логарифмически с увеличением размера
  - Элементы должны поддерживать операцию сравнения

Реализация ассоциативных списков

- Напоминает реализацию множеств
  - Ключи используются для поиска значений в списках
  - Ключи не повторяются
- HashMap
  - ключи хешируются
  - Быстрый поиск, но элементы хранятся в случайном порядке
- TreeMap
  - ключи сортируются
  - Поиск медленнее, но элементы хранятся в порядке сортировки.

Коллекции в Java 1.5

- До версии Java 1.4 элементы коллекций имели тип Object

```
LinkedList points = new LinkedList();
points.add(new Point(3, 5));
Point p = (Point) points.get(0);
```

  - Для объектов других типов всегда приходилось явно делать преобразование типа
  - В коллекцию элементов Point можно было добавить объекты других типов!
- В версии Java 1.5 добавлена поддержка т.н. обобщенного программирования (generics)
  - Класс или интерфейс может иметь параметр, указывающий на тип данных, с которыми будет работать создаваемый экземпляр этого класса/интерфейса:

```
LinkedList<Point> points = new
LinkedList<Point>();
points.add(new Point(3, 5));
Point p = points.get(0);
```

  - Преобразование типов больше не требуется
  - В такую коллекцию можно добавить только элементы типа Point
  - Это очень полезное расширение синтаксиса языка.

Применение коллекций

- Работа со списками и множествами не составляет большого труда:

```
HashSet<String> wordList = new HashSet<String>();
LinkedList<Point> waypoints = new LinkedList<Point>();
```

  - Заметим, что тип элементов следует указывать в типе переменной и в операторе new
- Инициализация ассоциативных списков чуть более многословна:

```
TreeMap<String, WordDefinition> dictionary = new
   TreeMap<String, WordDefinition>();
```

  - Сначала указывается тип ключа, затем тип значения
- Другие подробности смотрите в документации Java API

Перебор элементов коллекций

```
ArrayList – перебор делается очень просто:
ArrayList<String> quotes;
...
for (int i = 0; i < quotes.size(); i++)
    System.out.println(quotes.get(i));
```

- Этот способ нельзя использовать для коллекций с другим типом элементов
- Для перебора элементов коллекций есть другой универсальный метод - интерфейс Iterator:
  - hasNext() – возвращает true если имеется следующий элемент
  - next() – возвращает следующий элемент коллекции
- У интерфейса Interator имеется расширение ListIterator
  - Этот интерфейс имеет множество дополнительных функций.

Использование итераторов

- Итератор коллекции можно получить с помощью метода iterator()
- Пример:

```
HashSet<Player> players;
...
Iterator<Player> iter = players.iterator();
while (iter.hasNext()) {
    Player p = iter.next();
    ... // делаем что то с p
}
```

- Итераторы также используют технологию обобщенного программирования
- Итератор можно использовать для удаления текущего элемента

Расширенный синтаксис цикла for в Java 1.5

- Работа с итераторами по схеме которая использована в примере, не очень удобна
- Поэтому в Java 1.5 появилось расширение синтаксиса для упрощения работы с итераторами

```
for (Player p : players) {
    ... // делаем что то с p
}
```

  - Итератор используемый в цикле недоступен
  - Такой способ удобен для простого перебора элементов коллекции

- Расширенный синтаксис цикла for можно использовать с массивами:

```
float sum(float[] values) {
    float result = 0.0f;
    for (float val : values)
        result += val;
    return result;
}
```

Алгоритмы коллекций

- Класс java.utils.Collections имеет несколько полезных алгоритмов
    - Не путать с интерфейсом Collection
    - Реализованы в виде статических методов
    - Алгоритмы реализованы эффективно, имеют высокое быстродействие и используют синтаксис обобщенного программирования.
- Например , сортировка:

```
LinkedList<Product> groceries;
...
Collections.sort(groceries);
```

    - Метод не создает новую отсортированную коллекцию, изменяется порядок элементов коллекции groceries
- Смотрите подробности в документации Java API
    - Класс Array также имеет реализации полезных алгоритмов.

Элементы коллекций

- Элементы коллекций могут иметь некоторые дополнительные свойства
- Для элементов класса List ничего дополнительно не требуется
  - … если только не используются методы contains(), remove() и им подобные
    - В этом случае элементы должны иметь правильную реализацию метода equals()
- Такая правильная реализация должна удовлетворять условиям:
  - a.equals(a) == true
  - a.equals(b) == b.equals(a)
  - если a.equals(b) == true и b.equals(c) == true, то a.equals(c) == true
  - a.equals(null) == false

Элементы множеств и ассоциативных списков

- Элементы множеств и ассоциативных списков должны иметь дополнительные свойства
  - Множества должны поддерживать операции над элементами
  - Ассоциативные списки должны поддерживать те же операции над ключами
- Метод equals() должен работать правильно
- Элементы TreeSet, TreeMap должны поддерживать сортировку
  - Класс элемента должен содержать реализацию интерфейса java.util.Comarable или java.util.Comarator
- Элементы HashSet, HashMap должны поддерживать хеширование
  - Для них требуется реализация метода Object.HashCode().

Object.hashCode()

- Метод hashCode() объявлен в классе java.lang.Object:

  public int hashCode()

  - Он вычисляет хэш значения объекта
  - hashCode() используется классами HashSet, HashMap и некоторыми другими классами
- Правило 1:
  - Два одинаковых объекта (a.equals(b) == true) должны иметь одинаковые хэш коды
  - Но и два разных объекта могут иметь одинаковые хэш коды
    - То что объекты имеют одинаковые хэш коды означает только лишь, что сами они “могут быть одинаковыми”
- Правило 2:
  - Если вы в своем классе изменяете реализацию equals родительского класса, вам следует также сделать свою реализацию hashCode()
  - (См. правило 1)

Реализация hashCode()

- Рассмотрим такой вариант реализации этого метода:

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

  - Он удовлетворяет указанным выше правилам? Технически да …
  - но на практике он делает программу крайне неэффективной
- Функция вычисления хэша должна генерировать значения в широком диапазоне
  - Точнее говоря, значения этой функции должны принадлежать равномерному распределению
  - Это условие повышает эффективность выполнения операций над хэш таблицами
  - В примере выполнено основное требование к значению хэша - одинаковые объекты должны иметь идентичные значения хэша
  - Но также неплохо устроить так, чтобы неодинаковые объекты имели различные значения хэша.

Реализация hashCode() (2)

- Если поле класса включено в процедуру сравнения equals(), оно также должно участвовать в вычислении хэша
- Комбинируем несколько значений для вычисления хэша

```
int hashCode() {
    int result = 17; // какое то начальное значение
    // используйте другие начальные значения
    // для добавления полей в хэш
    result = 37 * result + field1.hashCode();
    result = 37 * result + field2.hashCode();
    ...
    return result;
}
```

Вычисление хэша

- Еще несколько рекомендаций
  - Если поле логическое (типа boolean) используйте числа 0 и 1 для хэш кода
  - Если поле имеет целый тип преобразуйте его к int
  - Если поле это объект (но не массив)
    - Вызовите его метод hashCode() или используйте 0 для null
  - Если поле массив
    - Добавляйте в хэш все его элементы
  - Подробнее см. Item 8 Effective Java: Programming Language Guide, Joshua Bloch, Addison Wesley, 2001, ISBN: 0-201-31005-8.
- Если вычисление хэша требует много времени, кэшируйте результат
  - В этом случае, пересчет хэша следует выполнять каждый раз при изменении объекта

Сравнение и упорядочивание объектов

- Интерфейс java.lang.Comparable<T> используется для сортировки элементов:

  public int compareTo(T obj)

- Метод возвращает значение, которое указывает на порядок следования элементов:
  - Результат < 0 означает, что this меньше чем obj
  - Результат == 0 означает, что this и obj одинаковы
  - Результат > 0 означает, что this больше чем obj
- Этот метод определяет естественный порядок следования экземпляров класса
  - то есть, “обычный” или “наиболее подходящий” порядок сортировки
- Порядок сортировки должен соответствовать реализации метода equals()
  - a.compareTo(b) возвращает 0 только если a.equals(b) == true
- Этот интерфейс следует реализовать для элементов TreeSet/TraaMap.

Другие способы сортровки

- Для сортировки можно использовать другой тип функций
  - Для этого нужен отдельный объект реализующий интерфейс java.util.Comparator<T> :

    int compare(T o1, T o2)
- Алгоритмы сортировки классов коллекций так же могут с помощью объекта компаратора сортировать свои элементы
  - Могут выполнять самые различные виды сортировки
- Как правило, компаратор делается как вложенный класс
  - Например, класс Player может иметь вложенный (объявленный внутри него) класс ScoreComparator

Java generics в интерфейсах

- Тип интерфейса java.lang.Comparable<T> :

  int compareTo(T obj)

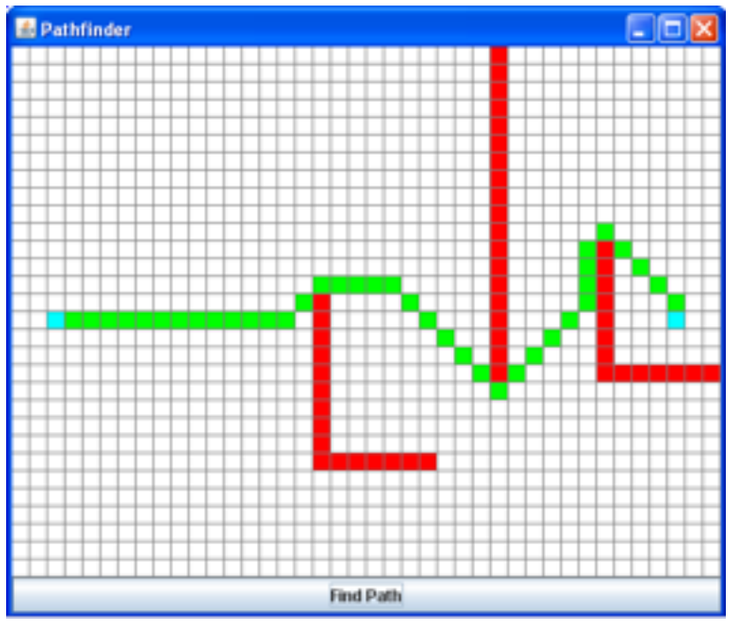
- Когда реализуете интерфейс, что такое T следует указать в коде:

```
class Player implements Comparable<Player> {
    ...
    int compareTo(Player obj) {
        ...
    }
}
```

- Эту же технологию можно использовать с интерфейсом java.util.Comparator.

Задание 2 - Поиск маршрута A*

- Алгоритм поиска маршрута A* широко используется в навигационных картах с препятствиями
  - Он позволяет найти оптимальный маршрут между двумя координатами, если маршрут существует
- Пример:



Реализация A*

- Для реализации алгоритма A* требуются две коллекции:
  - Коллекция “открытых точек маршрута” , которые предстоит проверить
  - Коллекция “закрытых точек маршрута” , которые уже проверены
- Ваши задачи:
  - Написать методы equals() и hashCode() для класса Loaction
  - Доделать класс AStarState, который контролирует открытые и закрытые точки маршрута A* алгоритма.
  - Поиграть с забавным пользовательским интерфейсом A* алгоритма ☺