Язык программирования Java

Лекция 3

Перевод курса CS11 Java Track

Содержание

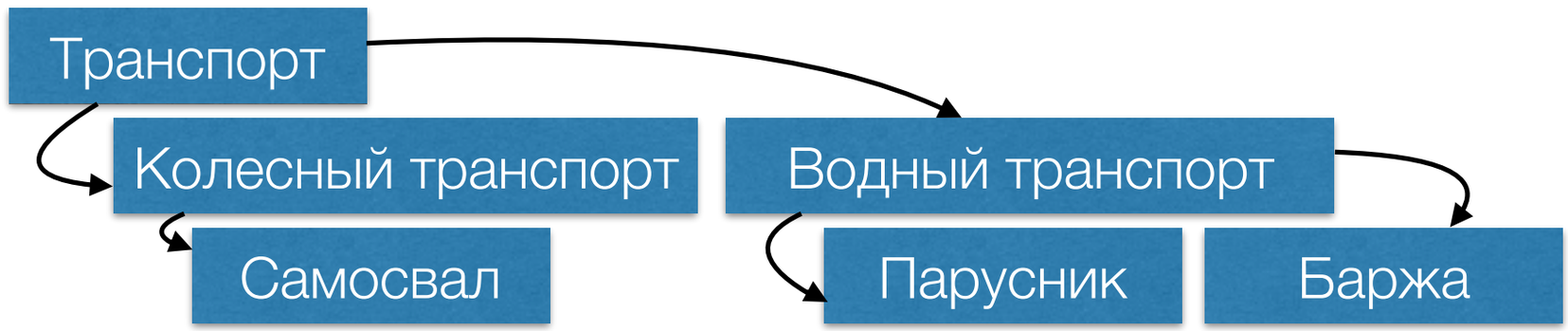
- Наследование
- Абстрактные классы
- Полиморфизм
- Введение в Swing API
- Вложенные и внутренние классы

Наследование

- Третья их четырех основных концепций объектно-ориентированного программирования
- Класс может расширять функционал другого класса
- Терминология:
  - Родительский класс, или базовый класс
  - Класс потомок, наследник, или производный класс
- Классы потомки могут наследовать все методы и поля родительского класса
  - Добавлять новый функционал
  - А также переопределять методы родительского класса

Наследование (2)

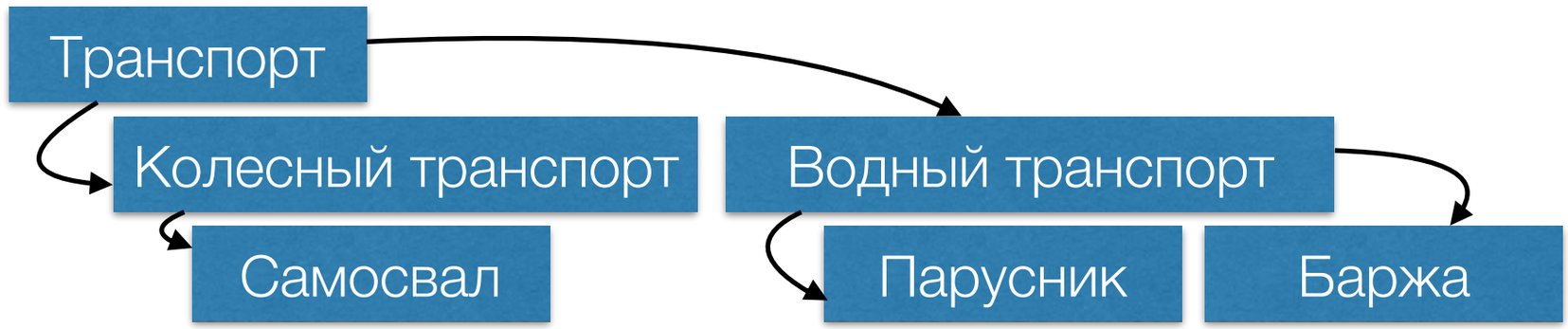
- Наследование моделирует отношение гипоним-гипероним (то есть более частного к более общему)
  - Пример иерархии классов:



- Класс наследник содержит уточнение свойств родительского класса
- Наследник имеет все свойства родительского класса
  - С наследником можно работать, считая его родительским классом:
    - “Самосвал это колесный транспорт”
    - “Парусник это транспорт”
    - “Водный транспорт это транспорт”

Наследование (3)

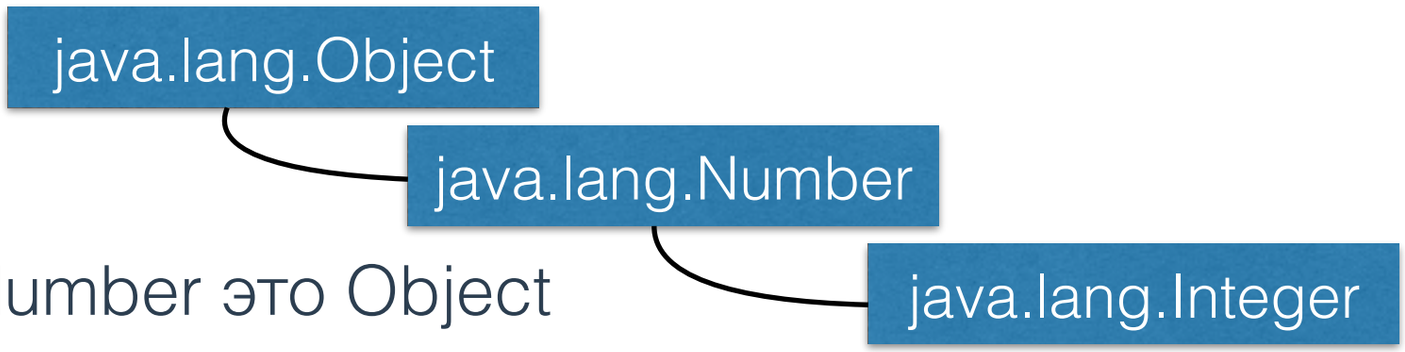
- Пример иерархии классов:



- Между классами в разных ветках иерархии отсутствует такая модель отношений
  - Очевидно, что следующие утверждения ошибочны:
    - “Самосвал это водный транспорт”
    - “Колесный транспорт это баржа”
- А как быть с утверждениями:
    - “Транспорт это самосвал”
    - “Водный транспорт это парусник”?
  - Зависит от того какое транспортное средство мы рассматриваем
    - Для того чтобы проверить эти утверждения надо изучить конкретное транспортное средство

Пример иерархии классов

- Классы чисел Java:



- Integer это Number, Number это Object
  - Класс Integer расширяет свойства класса Number, который в свою очередь расширяет свойства Object
- Integer наследует методы объявленные в Object:
  - boolean equals(Object o)
  - int hashCode()
  - String toString()
  - Class getClass()
- Класс Integer также переопределяет некоторые из этих методов

Переопределение Object.toString()

- Object.toString() весьма полезная мысль, особенно для отладки
- Этот метод также используется для соединения строк
  - Встретив такой код:

```
String msg = "Point is " + pt;
```

  - Компилятор автоматически заменит его на:

```
String msg = "Point is " + pt.toString();
```

- Переопределить метод не сложно:

```
@Override
public String toString() {
    return "(" + xCoord + "," + yCoord + ")";
}
```

Классы и объекты

- Родительские методы, в классе потомке, можно вызывать без какого либо особенного синтаксиса

```
Integer intObj = new Integer(53);
...
Class c = intObj.getClass(); // Получаем информацию о типе
```

  - Класс Integer это также и класс Object. Поэтому он может вызывать методы, объявленные или реализованные в классе Object.

- Класс потомок может также иметь свои собственные методы:

```
System.out.println(“Значение равно " + intObj.intValue());
```

  - Класс Integer расширяет функционал Object
    - intValue() возвращает int версию класса Integer

Ссылочные типы

- Любая ссылка имеет связанный с ней класс

```
Object obj; // ссылка на объект типа Object
Integer val; // ссылка на объект типа Integer
```

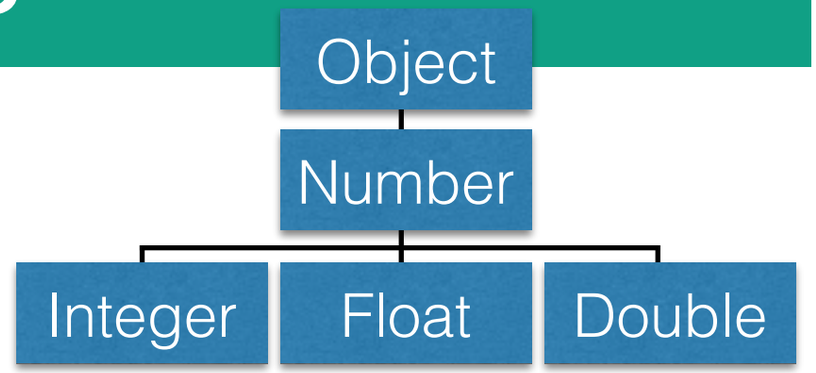
- Тип переменой определяет ее свойства
- Пример:

```
Object obj = new Integer(38);
...
System.out.println(obj.intValue()); ОШИБКА КОМПИЛЯЦИИ
```

  - Причина ошибки компиляции в том, что Object не имеет метода intValue()
  - Этот метод объявлен в классе Number (родительский класс Integer)
  - Даже если obj указывает на экземпляр класса Integer, видны будут только методы класса Object

Перемещение по иерархии классов

- Иерархия чисел выглядит так:



- Для того чтобы переместиться вниз по иерархии требуется выполнить проверку во время исполнения кода:

```
Object obj = new Integer(453);
...
int i = ((Integer) obj).intValue(); // преобразование типа obj
```

  - Можно попробовать и так:

```
float f = ((Float) obj).floatValue(); // Ошибка времени исполнения
```

    - Этот код будет компилироваться, но выдаст ошибку во время исполнения.
  - Java не может определить действительный тип переменной во время компиляции
    - (даже если это очевидно для человека)
  - Проверка происходит во время исполнения и может привести к ошибке

Что не могут классы потомки

- Классы потомки не могут обращаться к членам родительского класса, которые имеют модификатор доступа private
- Модификатор protected разрешает обращение к членам родительского класса
  - Члены класса доступны из самого класса и из его потомков
  - Защита менее строгая, чем private, но это все же не public
- Дочерние классы не могут наследовать статические поля и методы
  - Но они остаются доступными

Пример. Класс Task

```
public class Task {
    private String name;
    private boolean done;
    public Task(String taskName) {
        name = taskName;
        done = false;
    }
    /** Отмечает что задача завершена. */
    public void doTask() {
        done = true;
    }
    /** Возвращает признак завершения задачи. */
    public boolean isDone() {
        return done;
    }
}
```

Добавляем задание

- Класс Task это базовый класс
    - … настолько базовый, что он почти бесполезен
    - Расширим его функции:

```
public class FileUploadTask extends Task {
    public FileUploadTask() {
        // вызов конструктора родительского класса
        super("upload file");
    }
    ...
}
```

- Конструктор родительского класса не наследуется!
- Если у родительского класса нет конструктора по умолчанию, необходимо явно вызвать его конструктор в конструкторе дочернего класса с помощью ключевого слова super

Переопределение методов родительского класса

- Класс FileUploadTask должен иметь собственную реализацию метода doTask()

```
public class FileUploadTask extends Task {
    ...
    /** Выполняются операции загрузки файла. */
    @Override
    public void doTask() {
        ... // открываем соединение, считываем файл, и т.д.
    }
}
```

- Сигнатура метода должна совпадать с сигнатурой этого метода в родительском классе
- Это переопределяет реализацию метода doTask() в классе Task

Полиморфизм

- Теперь нам требуется загрузить файл:

```
Task t = new FileUploadTask();
t.doTask();
```

  - В какой реализации doTask() следует поместить это вызов?

- В Java все нестатические методы виртуальные
  - Даже если t это ссылка на экземпляр класса Task, будет вызвана реализация метода FileUploadTask
  - Это происходит потому, что t в действительности указывает на объект типа FileUploadTask
- Это свойство называется полиморфизм
  - Четвертая основная концепция ООП
  - Действие, которое выполняется этими строками кода, зависит от типа объекта (а не от типа ссылки)

Вызов методов родительского класса

- Проблема:
  - FileUploadTask.doTask() не устанавливает done в true
  - К тому же done имеет модификатор private
- Одно из возможных решений:
  - FileUploadTask.doTask() может вызвать свою реализацию в родительском классе:

```
/** Выполняет загрузку файла. */
@Override
public void doTask() {
    ... // открываем соединение, считываем файл, и т.д.
    // Готово!
    super.doTask();
}
```

Абстракция Task

- Реализация doTask() для класса Task не имеет смысла
  - Сделаем класс Task абстрактным
  - Абстрактный класс содержит только объявление поведения без его определения, то есть без реализации
- Нельзя создавать экземпляры абстрактных классов
  - Абстрактные классы должны иметь наследников, которые реализуют недостающий функционал
  - Пример: FileUploadTask должен обеспечить реализацию метода doTask(), который загружает файл

Новый абстрактный класс Task

- Абстрактный класс Task:

```
//Общий класс задач
public abstract class Task {
    private String name;
    private boolean done;
    public Task(String taskName) {
        name = taskName;
        done = false;
    }
    //Этот метод следует реализовать в классах наследниках.
    public abstract void doTask();
    ... // Остальной код класса
}
```

- Абстрактные классы могут иметь поля и неабстрактные методы

Новый класс FileUploadTask

- Класс FileUploadTask не ‘переопределяет' метод doTask()
  - Нечего переопределять!
  - FileUploadTask *реализует* doTask()
- Сигнатуры методов, по прежнему, должны совпадать:

```
/** Реализация doTask() для загрузки файла. */
public void doTask() {
    ... // открываем соединение, считываем файл, и т.д.
}
```

  - (Конечно без модификатора abstract!)
  - Теперь уже нельзя вызвать super.doTask()
- Класс наследник должен иметь реализацию всех абстрактных методов родительского класса
  - Если это не так класс тоже должен быть объявлен абстрактным.

Завершение абстракции

- Как указать, что задача выполнена?
- Проще всего изменить модификатор поля done на protected
- Еще один хороший способ: добавить к классу Task еще один protected метод:

```
protected void reportTaskDone() {
    if (done) {
        ... // Задача уже завершена!.
    }
    done = true;
}
```

  - Теперь только классы потомки могут сообщать об окончании задания
- Какое из этих решений лучше “расширяется”?
  - Может возникнуть необходимость выполнить какую либо дополнительную обработку, когда задача завершается
  - Это можно будет легко потом добавить в reportTaskDone()

Ссылки на Task

- Экземпляр абстрактного класса Task создать нельзя

```
Task t = new Task("send e-mail"); ОШИБКА КОМПИЛЯЦИИ
```

  - Так как реализация класса Task не завершена
- Но со ссылкой на класс Task можно работать

```
Task t = new FileUploadTask();
t.doTask(); // вызов FileUploadTask.doTask()
t = new SendEMailTask();
t.doTask(); // вызов SendEMailTask.doTask()
```

  - Правильная реализация doTask() будет вызвана благодаря полиморфизму
- API становятся общими для всех классов в цепочке наследования, если используют ссылки на базовый класс:

```
void enqueueTask(Task t) {
    pendingList.store(t);
}
```

Swing краткий обзор

- Первая программная оболочка для создания пользовательского интерфейса Java называлась AWT
  - Abstract Windowing Toolkit
  - Эта оболочка имела набор базовых функций
  - Выглядела не очень красиво и была не расширяемой
- В версии Java 1.2 появился Swing API
  - Этот интерфейс построен на некотором функционале AWT
  - Многие высокоуровневые классы AWT были переделаны
  - Внешний вид стал перенастраиваемым
  - Интерфейс стал расширяемым и предлагает большой набор функций.
  - Однако Swing работает медленнее, чем AWT так как он “полностью реализован на Java”

Классы Swing

- Большинство классов Swing располагаются в пакете javax.swin (и некоторых производных от этого пакетах)
- Swing использует довольно мало классов AWT
  - События, обработчики событий, геометрические функции, изображения, drag-drop и др.
- Виджеты пользовательского интерфейса Swing наследуют класс JComponent
  - Этот класс базовый для всех пользовательских компонентов Swing
  - Класс JComponent наследник класса java.awt.Container.
  - Пользовательские компоненты Swing могут наследоваться от JComponent

Тяжеловесные компоненты

- Компоненты пользовательского интерфейса AWT “тяжеловесные” они требуют много ресурсов для работы
- Каждый компонент имеет свой собственный набор графических ресурсов
- Компоненты не используют только Java для прорисовки графики
  - Для этого используются системные API функции
- Перекрывающиеся на экране компоненты перерисовываются друг на друге

Легковесные компоненты

- Компоненты Swing напротив “лекговесны”
  - Они используют для собственной прорисовки только код Java
  - Графические ресурсы платформы совместно используются Swing компонентами насколько это возможно
- Пример:
  - Всплывающие меню внутри окна приложения прорисовываются с помощью ресурсов этого окна
  - Всплывающее меню за пределами окна приложения получает свое собственное окно
- Swing эффективно работает с прозрачными областями, так как компоненты совместно используют графические ресурсы

Совмещение AWT и Swing

- “Легковесные” и “тяжеловесные” компоненты трудно совмещаются
  - “Тяжеловесные” компоненты всегда прорисовываются поверх “легковесных”
- Избегайте по возможности одновременно использовать в программе компоненты AWT и Swing

Окна и контейнеры

- JWindow – простое окно
  - … но, без заголовка, меню и стандартных кнопок
- JFrame – окно приложения
  - С меню, заголовком и кнопками
  - Обычно используется в Java приложениях с графическим пользовательским интерфейсом
- JPanel – группирует вместе компоненты пользовательского интерфейса
  - “Легковесный” контейнер общего назначения
  - Хорошо подходит для создания структуры вашего пользовательского интерфейса
- Для добавления дочерних компонентов используется метод add()
  - Дочерние элементы также могут быть контейнерами, как например JPanel.

Расположение компонентов

- Контейнеры позиционируют и устанавливают размер дочерних компонентов с помощью контроллеров расположения
  - Для этого вызывается метод контейнера setLayout(LayoutManager lm)
  - Интерфейс java.awt.LayoutManager
- Есть много разных контроллеров расположения:
  - FlowLayout – размещает компоненты друг за другом в строке, переходя на следующую строку, когда в текущей не остается места.
  - BoxLayout - размещает компоненты в одной строке или колонке.
  - BorderLayout – может поместить компонент в одну из пяти областей: NORTH, SOUTH, EAST, WEST и CENTER.
  - GridLayout – размещает компоненты в двумерной таблице.
  - GridBagLayout – контролер расположения имеющий расширенных функционал
  - Есть и другие контроллеры (см. LayoutManager)
- По умолчанию используется контроллер FlowLayout

События и обработчики событий

- Виджеты пользовательского интерфейса генерируют события
  - Когда пользователь кликает мышью
  - Нажимает клавишу на клавиатуре
  - Перемещает курсор мыши или перетаскивает мышью объект
  - Когда закрывается или минимизируется окно
  - И во многих других случаях.
- Для обработки событий в программу добавляются процедуры обработчики событий
  - Обработчики описываются интерфейсами
  - Интерфейсы объявленны в java.awt.event
  - Обычно имя обработчика заканчивается на […]Listener.

Интерфейс ActionListener

- Пример: интерфейс java.awt.event.ActionListener
  - Имеет один метод:

    void actionPerformed(ActionEvent e)

- ActionEvent содержит детали произошедшего события:
  - Компонент источник события
  - Время, когда оно произошло
  - Состояние служебных клавиш (Ctrl, Alt, Shift, …)
  - Другую информацию о которой можно прочитать в документации
- Большинство компонентов Swing генерирует событие ActionEvent

Реализация ActionListener

- Для регистрации обработчика этого события используется метод

  addActionListener(ActionListener l)

- Реализация ActionListener выглядит так:

```
public class ActionHandler implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
        ... // Делаем что то полезное.
    }
}
```

- Регистрация обработчика:

```
ActionHandler handler = new ActionHandler();
JButton button = new JButton("Старт");
button.addActionListener(handler);
```

Другие интерфейсы обработчиков AWT/Swing

- MouseListener – события о попадании в окно, выходе за пределы окна и нажатии кнопок мыши.
- MouseMotionListener – сообщения о перемещении курсора мыши или перетаскивании мышью объектов
- KeyListener – нажатие и отпускание клавиш клавиатуры
- FocusListener – компонент получает/теряет фокус
- ComponentListener – компонент меняет размер, делается видимым или невидимым
- WindowListener – окно открывается, закрывается, расширяется до размеров экрана или минимизируется

Обработчики и адаптеры

- Некоторые интерфейсы обработчиков излишне сложны:
  - MouseListener наример, содержит следующий набор методов:
    - mouseEntered(), mouseExited()
    - mousePressed(), mouseReleased()
    - mouseClicked().
- Часто из всего списка требуется реализовать только один или два метода
- Java предлагает адаптеры для интерфейсов обработчиков событий
- Пример: java.event.MouseAdapter
  - Содержит реализацию интерфейса MouseListener
  - Как и требуется, реализованы все функции интерфейса
  - Создайте свой класс наследник MouseAdapter и переопределите в нем только нужные вам методы обработчики

Вложенные классы Java

- В языке Java можно объявлять один класс внутри другого
  - Такой класс называется вложенным:

```
class Outer {
    /* Вложенный класс */
    class Inner {
        ...
    }
}
```

- Компилятор из файла Outer.java создаст два файла: Outer.class и Outer$Inner.class.

Вложенные классы Java (2)

- Вложенный класс является членом внешнего класса и может иметь модификатор доступа
  - Например, вложенный класс с модификатором private не может быть использован извне класса, в котором он объявлен
- Вложенный класс может иметь или не иметь модификатор static
  - Модификатор сильно влияет на правила работы с классом:

```
class Outer {
    static class StaticNested { ... }
    class NonStaticNested { ... }
}
```

Статические вложенные классы

- Статические вложенные классы это просто классы логически связанные со своим внешним классом
- Пример: java.awt.geom.Rectangle2D
  - Абстрактный класс двухмерных прямоугольников
- Содержит два статических вложенных класса:
  - Rectangle2D.Double наследника Rectangle2D, который использует тип double для хранения значений координат
  - Rectangle2D.Float который использует для этой цели тип float
- Для того чтобы использовать эти классы
  - import java.awt.geom.Rectangle2D;
  - Обращаться к вложенным классам Rectangle2D.Float или Rectangle2D.Double.

Не статические вложенные классы

- Не статические вложенные классы также называются внутренними классами
- Также как нестатические методы, внутренние классы должны использоваться в контексте содержащего их класса
  - Они имеют доступ к объекту содержащего их класса
  - Могут обращаться к его полям и методам
- Объекты внутренних классов нельзя создавать в статических методах внешнего класса
  - Только в методах экземпляра класса (т.е. нестатических)!

Обработчики событий во внутренних классах

- Внутренние классы удобно использовать для обработки событий
  - Обработчикам событий, как правило, нужен доступ к состоянию приложения
  - Внутренние классы могут обращаться даже к private членам внешнего класса
- Использование внутренних классов позволяет убрать “лишние” public методы обработчиков событий внешнего класса
  - Внешний класс не обязан показывать наружу все имеющиеся у него методы интерфейсов обработчиков событий
- При необходимости можно создать несколько объектов внутреннего класса связанных с одним экземпляром внешнего класса

Обработчики событий во внутренних классах

```
public class MyApp {
    /** Текущее состояние приложения. **/
    private boolean started;
    /** Обработчик ActionEvents. **/
    private class ActionHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            started = true;
        }
    }
    ...
    void initUI() {
        // Создаем копку и внутренний класс для обработки ее событий
        JButton button = new JButton("Start");
        button.addActionListener(new ActionHandler());
    }
}
```