**Обзор Java**

Приложения Android пишутся в основном на языке Java. Это приложение содержит обзор некоторых основ Java, необходимых при разработке для Android, представленных в виде набора практических упражнений.

Код этих упражнений можно найти по адресу <https://github.com/info448/appendix-java-review> .

**A.1 Создание приложений с помощью Gradle**

Рассмотрим включенный Dogкласс, найденный в src/main/java/edu/info448/review/папке. Это очень простой класс, представляющий Собаку. Вы можете создавать экземпляры и вызывать методы этого класса, создавая и запуская Testerкласс, найденный в той же папке. - Вы можете просто использовать любой текстовый редактор, например *VS Code* , *Atom* или *Sublime Text,* чтобы просматривать и редактировать эти файлы.

Вы, вероятно, запускали программы Java с помощью IDE, но давайте рассмотрим, что входит в создание этого приложения «вручную» или просто с использованием инструментов JDK. Существует два основных шага для запуска программы Java:

1. **Компиляция.** Это преобразует исходный код Java (в .javaфайлах) в байт-код JVM, понятный виртуальной машине (в .classфайлах).
2. **Запуск** Фактически это загружает байт-код в виртуальную машину и выполняет main()метод.

Компиляция выполняется с помощью javacкоманды («java compile»). Например, из каталога репозитория кода вы можете скомпилировать оба .javaфайла с помощью:

*# Compile all .java files*

javac src/main/java/edu/info448/review/\*.java

Затем запуск выполняется с помощью javaкоманды: вы указываете полное имя пакета класса, который хотите запустить, а также [путь к классам](https://docs.oracle.com/javase/tutorial/essential/environment/paths.html) , чтобы Java знала, где искать классы, от которых он зависит:

*# Runs the Tester#main() method with the `src/main/java` folder as the classpath*

java -classpath ./src/main/java edu.info448.review.Tester

***Практика: Скомпилируйте и запустите это приложение прямо сейчас.***

***Практика: Измените Dogкласс так, чтобы его .bark()метод лаял дважды ( "Bark Bark!"). Что нужно сделать, чтобы проверить, что ваши изменения сработали?***

Вы можете заметить, что этот цикл разработки может быть довольно утомительным: для запуска нашего кода нам нужно выполнить две команды, и обе они настолько сложны, что их сложно перепечатывать.

Введите [**Gradle**](https://gradle.org/) . Gradle — это система автоматизации сборки: «скрипт», который вы можете запустить, чтобы автоматически выполнить несколько шагов, необходимых для сборки и запуска приложения. Этот скрипт определяется файлом build.gradleконфигурации. ***Практика: откройте этот файл и просмотрите его содержимое*** . task run()Здесь определена задача «запустить»: видите, как она определяет те же аргументы, которые мы в противном случае передали команде java?

Вы можете запустить версию Gradle, включенную в репозиторий, с помощью gradlew <task>команды, указав, какую задачу должна выполнить система сборки. Например:

*# on Mac/Linux*

./gradlew tasks

*# on Windows*

gradlew tasks

Даст вам список доступных задач. Используйте gradlew classesдля компиляции кода, а также gradlew runдля компиляции *и* запуска кода.

* **Полезный совет** : вы можете указать флаг «quite», gradlew -q <task>чтобы Gradle не выводил свой статус сборки (удобно для этой runзадачи)

***Практика: Используйте gradle для сборки и запуска программы Dog. Видите, насколько это проще?***

Мы будем использовать Gradle для создания наших приложений Android (которые намного сложнее, чем эта простая демонстрация Java)!

**A.2 Основы класса**

Теперь рассмотрим Dogкласс более подробно. Как и все классы, он состоит из двух частей:

1. **Атрибуты** (они же переменные экземпляра, поля или переменные-члены). Например, String name.
   * Обратите внимание, что все эти атрибуты являются private, то есть они не доступны членам другого класса! Это важно для **инкапсуляции** : это означает, что мы можем изменить способ Dogреализации класса, не изменяя никакой другой класс, который зависит от него (например, если мы хотим хранить breedкак число вместо String).
2. **Методы** (они же функции). Напримерbark()
   * Обратите внимание *на объявление метода* public void wagTail(int) . Эта комбинация модификатора доступа ( public), возвращаемого типа ( void), имени метода ( wagTail) и параметров ( int) называется **сигнатурой метода** : это «автограф» этого конкретного метода. Когда мы вызываем метод (например, myDog.wagTail(3)), Java будет искать определение метода, которое *соответствует* этой сигнатуре.
   * Сигнатуры методов очень важны! Они говорят нам, какими будут входы и выходы метода. Мы должны понимать, как работает метод, *просто* из его сигнатуры.

Обратите внимание, что один из методов .createPuppies()— это staticметод. Это означает, что метод принадлежит классу ***,*** а не отдельным экземплярам объектов класса! ***Практика: попробуйте запустить следующий код (поместив его в main()метод класса Tester)*** :

Dog[] pups = Dog.createPuppies(3);

System.out.println(Arrays.toString(pups));

Обратите внимание, что для вызова createPuppies()метода вам не нужно иметь Dogобъект (вам не нужно использовать newключевое слово): вместо этого вы обращаетесь к «шаблону» для Dogи говорите этому шаблону выполнить некоторую работу. *Нестатические* методы (те, у которых нет staticключевого слова, также называемые «методами экземпляра») должны вызываться для объекта.

***Практика: Попробуйте запустить код Dog.bark(). Что происходит?*** Это потому, что вы не можете сказать «шаблону» для aDogлаять, только реальномуDogобъекту!

В общем, в 98% случаев ваши методы **не** должны быть static, потому что вы хотите вызывать их для конкретного объекта, а не для общего «шаблона» для объектов. Переменные **никогда не** должны быть статическими, если только они не являются **также** final константами (как BEST\_BREEDпеременная).

* В Android staticпеременные вызывают значительные утечки памяти, а также в целом являются неудачным решением.

**А.3 Наследование**

***Практика: Создайте новый файл Husky.java, объявляющий новый Huskyкласс:***

**package** edu.info448.review; *//package declaration (needed)*

**public** **class** Husky **extends** Dog {

*/\* class body goes here \*/*

}

Ключевое extendsслово означает, что это Huskyподкласс [**,**](https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html)Dog наследующий все его методы и атрибуты. Это также означает, что экземпляр Huskyявляется **экземпляром** Dog .

***Практика: В Tester создайте экземпляр new Huskyи вызовите bark()его. Что произойдет?***

* Поскольку мы унаследовали от Dog, Huskyкласс получает все методы, определенные в , Dogбесплатно!
* Попробуйте добавить конструктор, который принимает один параметр ( name) и вызывает соответствующий super()конструктор, чтобы breed был "Husky", что делает это немного более разумным.

Мы также можем добавить больше методов в **подкласс** , которых нет в **родительском классе . *Практика: добавьте метод, вызываемый .pullSled()в Huskyкласс.***

* Попробуйте вызвать .pullSled()свой Husky *объект . Что* происходит? Затем попробуйте вызвать .pullSled()объект . Что происходит?Dog

Наконец, мы можем **переопределить** методы из родительского класса. ***Практика: добавьте bark()метод в Husky(с той же сигнатурой), но с Husky«гав» вместо «барк».*** Протестируйте свой код, вызвав метод в Tester.

**А.4 Интерфейсы**

***Практика: Создайте новый файл Huggable.javaсо следующим кодом:***

**package** edu.info448.review;

**public** **interface** Huggable {

**public** void hug();

}

Это пример интерфейса [**.**](https://docs.oracle.com/javase/tutorial/java/concepts/interface.html) Интерфейс **—** это список методов, которые класс *обещает* предоставить. *Реализуя* интерфейс (с interfaceключевым словом в объявлении класса), класс обещает включить любые методы, перечисленные в интерфейсе.

* Это очень похоже на то, как если бы вы повесили снаружи своего заведения вывеску *«Принимаем Visa»* . Это значит, что если кто-то подойдет к вам и попытается заплатить картой Visa, вы сможете это сделать!
* Реализация интерфейса не дает никаких обещаний относительно *того, что* делают эти методы, а только то, что класс будет включать методы с этими сигнатурами. ***Практика: измените Huskyобъявление класса*** :

java public class Husky extends Dog implements Huggable {...}

Теперь класс Huskyдолжен иметь public void hug()метод, но что именно *делает* этот метод , решать вам!

* Класс может иметь .hug()метод даже без реализации Huggableинтерфейса (см. TeddyBear), но мы получаем больше преимуществ, объявляя о поддержке этого метода.
  + Точно так же, как вывеска «Принимаем Visa» привлечет больше людей, которые захотят расплачиваться кредитной картой, а не просто предоставят такую ​​возможность, если кто-то об этом спросит.

Почему бы просто не создать Huggableсуперкласс и не расширить Huskyего?

* Потому что Huskyextends Dog, а в Java может быть только один родительский элемент!
* А поскольку не все собаки Huggableи не все Huggableвещи являются таковыми Dogs, то не существует четкой иерархии того, куда включать интерфейс.
* Кроме того, мы можем реализовать несколько интерфейсов ( Husky implements Huggable, Pettable), но мы не можем наследовать от нескольких классов.
  + Это отлично подходит, когда у нас есть другие классы разных типов, но со схожим поведением: например, a TeddyBearможет быть Huggable, но не может bark()быть Dog!
  + ***Практика: Создайте класс TeddyBearimplement Huggable. Нужно ли добавлять какие-либо новые методы?***

***В чем разница между наследованием и интерфейсами?*** Главное практическое правило: используйте*наследование* (extends), когда вы хотите, чтобы классы разделяли**код** (реализацию). Используйте*интерфейсы* (implements), когда вы хотите, чтобы классы разделяли**поведение** (сигнатуры методов). В конце концов,*интерфейсы* важнее для хорошего объектно-ориентированного проектирования. Отдавайте предпочтение интерфейсам, а не наследованию!

**А.5 Полиморфизм**

Реализация интерфейса также устанавливает отношение **is** : так что Huskyобъект **является** Huggable объектом. Это позволяет получить наибольшее преимущество интерфейсов и наследования: [**полиморфизм**](https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html) , или возможность рассматривать один объект как тип другого!

Рассмотрим стандартное объявление переменной:

Dog myDog; *//= new Dog();*

Тип переменной myDog— Dog, что означает, что переменная может ссылаться на любое значение (объект), **являющееся** Dog .

***Практика: Попробуйте следующие объявления (обратите внимание, что некоторые из них не скомпилируются!)***

Dog v1 = **new** Husky();

Husky v2 = **new** Dog();

Huggable v2 = **new** Husky();

Huggable v3 = **new** TeddyBear();

Husky v4 = **new** TeddyBear();

Если **значение** (то, что справа) *является* экземпляром **типа переменной** (тип слева), то у вас есть допустимое объявление.

Даже если вы объявите переменную Dog v1 = new Husky(), **значением** в этом объекте *будет* a Husky. Если вы вызовете .bark()его, вы получите Huskyверсию метода ( ***Практика: попробуйте переопределить метод, чтобы вывести его "barks like a Husky"и увидеть*** ).

Вы можете **выполнять приведение** между типами, если вам нужно преобразовать один в другой. Пока **значение***является* экземпляром типа, к которому вы приводите, операция будет работать нормально.

Dog v1 = **new** Husky();

Husky v2 = (Husky)v1; *//legal casting*

Наибольшее преимущество полиморфизма — это абстракция. Рассмотрим:

ArrayList<Huggable> hugList = **new** ArrayList<Huggable>(); *//a list of huggable things*

hugList.add(**new** Husky()); *//a Husky is Huggable*

hugList.add(**new** TeddyBear()); *//so are Teddybears!*

*//enhanced for loop ("foreach" loop)*

*//read: "for each Huggable in the hugList"*

**for**(Huggable thing : hugList) {

thing.hug();

}

***Практика: Что произойдет, если запустить приведенный выше код?*** Поскольку хаски и плюшевые мишки ведут себя одинаково (interface), мы можем рассматривать их как один «тип» и, таким образом, поместить их обоих в список. И поскольку все в списке поддерживает интерфейсHuggable, мы можем вызвать.hug()каждый элемент в списке, и мы знаем, что у них будет этот метод —implementingв конце концов, они обещали интерфейс!

**A.6 Абстрактные методы и классы**

Еще раз взгляните на Huggableсозданный вами интерфейс. Он содержит единственное объявление метода… за которым следует точка с запятой вместо тела метода. Это [**абстрактный метод**](https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html) : на самом деле, вы можете добавить abstractключевое слово к этому объявлению метода, ничего не меняя (все методы являются интерфейсами неявно abstract, поэтому это не требуется):

**public** **abstract** void hug();

Абстрактный **метод** — это метод, у которого (пока) нет тела метода: это просто сигнатура, но нет фактической реализации. Он «незавершен». Чтобы создать экземпляр класса (используя newключевое слово), этот класс должен быть «завершен» и предоставить реализации для *всех* абстрактных методов — например, всех тех, которые вы унаследовали от интерфейса. Это именно то, что вы использовали interfacesдо сих пор: это просто другой способ думать о том, почему вам нужно предоставлять эти методы.

Если abstractключевое слово подразумевается для интерфейсов, в чем смысл? Рассмотрим Animalкласс (который является родительским классом для Dog). .speak()Метод «пустой»; чтобы он мог что-то сделать, подкласс должен его переопределить. И в настоящее время нет ничего, что могло бы помешать тому, кто создает подкласс, Animalзабыть реализовать этот метод!

Мы можем *заставить* подкласс переопределить этот метод, сделав метод abstract: фактически, оставив его незавершенным, так что если подкласс (например, Dog) захочет что-то сделать, он должен будет завершить метод. ***Практика: Создайте Animal#speak()метод abstract. Что произойдет, когда вы попытаетесь построить код?***

Если Animalкласс содержит незаконченный ( abstract) метод… то этот класс сам по себе незаконченный, и Java требует, чтобы мы отметили его как таковой. Мы делаем это, объявляя *класс* как abstractв объявлении класса:

**public** **abstract** **class** MyAbstractClass {...}

***Практика: Создайте Animalкласс abstract.*** Вам нужно будет предоставить реализацию метода.speak()вDogклассе: попробуйте просто вызвать.bark()метод (композиция методов для победы!).

Только абстрактные классы и interfacesмогут содержать abstractметоды. Кроме того, abstractкласс не является законченным, то есть его нельзя инстанцировать. ***Практика: попробуйте инстанцировать new Animal(). Что происходит?*** Абстрактные классы отлично подходят для содержания «большинства» класса, но при этом необходимо убедиться, что он не используется без всех предоставленных деталей. И если вы подумаете об этом, мы бы никогда не захотели инстанцировать обобщенный класс Animalв любом случае — вместо этого мы бы создали Dogили или Catили Turtleчто-то еще. Все, что Animalделает класс, — это действует как **абстракция** для этих других классов, чтобы позволить им совместно использовать реализации (например, метода walk()).

* Абстрактные классы немного похожи на «шаблоны» для классов… которые сами по себе являются «шаблонами» для объектов.

**A.7 Дженерики**

Говоря о шаблонах: вспомните класс ArrayList, который вы использовали в прошлом, и как вы указали «тип» внутри этого списка с помощью угловых скобок (например, ArrayList<Dog>). Эти угловые скобки указывают, что ArrayListэто [универсальный](https://docs.oracle.com/javase/tutorial/java/generics/) класс: шаблон для класса, где *тип данных* для этого класса сам по себе является переменной.

Рассмотрим GiftBoxкласс, представляющий собой коробку, содержащую TeddyBear. ***Какие изменения вам нужно будет внести в этот класс, чтобы он содержал Huskyвместо TeddyBear? А что, если Stringвместо этого он содержал ?***

Вы должны заметить, что единственное различие между TeddyGiftBoxи HuskyGiftBoxи StringGiftBoxбудет заключаться в **типе переменной** содержимого. Поэтому вместо того, чтобы дублировать работу и писать один и тот же код для каждого типа подарка, который мы хотим подарить… мы можем использовать **generics** .

Обобщения позволяют нам указывать тип данных (например, что в данный момент является TeddyBearили String) как *переменную* , которая устанавливается при создании экземпляра класса с помощью угловых скобок (например, new GiftBox<TeddyBear>()создаст объект класса с этой переменной типа, установленной как TeddyBear).

Мы указываем дженерики, объявляя переменную типа данных в объявлении класса:

**public** **class** GiftBox<T> {...}

( T— это общее имя переменной, сокращение от «Тип». Другие варианты включают Eэлементы в списках, Kключи и Vзначения в картах).

И затем везде, где вы бы поместили тип данных (например, TeddyBear), вы можете просто поместить Tпеременную вместо этого. Это будет заменено *фактическим* типом **во время компиляции** .

* Предупреждение: *всегда* используйте однобуквенные имена переменных для универсальных типов! Если вы попытаетесь назвать его как-то вроде String(например, public class GiftBox<String>), то Java будет интерпретировать слово Stringкак тип переменной, а не как ссылку на java.lang.Stringкласс. Это очень похоже на объявление переменной int Dog = 448, а затем вызов Dog.createPuppies().

***Практика: Попробуйте сделать GiftBoxкласс универсальным и создать новый экземплярGiftBox<Husky>***

**A.8 Вложенные классы**

И последнее: мы помещали *атрибуты* и *методы* в классы... но мы также можем определять дополнительные *классы* внутри класса! Они называются [**вложенными** или **внутренними классами**](https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html) .

Мы часто вкладываем «вспомогательные классы» внутрь более крупного класса: например, вы можете поместить Nodeкласс внутрь LinkedListкласса:

**public** **class** LinkedList {

*//nested class*

**public** **class** Node {

**private** int data;

**public** Node(int data) {

**this**.data = data;

}

}

**private** Node start;

**public** LinkedList() {

**this**.start = **new** Node(448);

}

}

Или, может быть, мы хотим определить Smellкласс внутри Dogкласса, чтобы представлять различные запахи, что позволит нам говорить о различных Dog.Smellобъектах. (И, конечно, Dog.Smellкласс будет реализовывать Sniffableинтерфейс…)

Вложенные классы, которые мы определяем, обычно static: то есть они принадлежат классу, *а* не экземплярам объектов этого класса. Это означает, что в памяти есть только одна копия этого вложенного класса чертежа; это эквивалентно размещению класса в отдельном файле, но вложение позволяет нам хранить их в одном месте и предоставляет функцию «пространства имен» (например, Dog.Smellа не просто Smell).

С другой стороны, нестатические вложенные классы (или **внутренние классы** ) определяются для каждого объекта. Это важно только в том случае, если поведение этого класса будет зависеть от объекта, в котором он находится. Это тонкий момент, который мы увидим, когда предоставим внутренние классы, требуемые фреймворком Android.