# Обобщенные типы

- Один из механизмов обобщения - в аргументе объект базового класса, после использование методов для производных классов (исп базового типа для повышения гибкости)
- Если аргумент метода - интерфейс, то использоваться могут все классы, наследующие его
- Для еще большего расширение можно сказать, что код работает с общим типом
- Обобщение - реализует концепцию параметризованных типов, (позволяет создать компоненты - контейнеры, используемые с разными типами) - применим
- При создании экземпляра параметризованного типа, все приведения типов за вас, правильность во
- время компиляции
- Созданные обобщения не оч
- Обобщение - больше про плюсы, но не про джаву
- Простые обобщения
- Одна из причин для применения - классы контейнеров. В контейнерах хранят используемые объекты
- В контейнере - один тип, причина обобщения - возможноть указать тип (раньше можно было использовать
- Object)
- Вместо него желательно использовать условный тип. Параметр-тип указывается в угловых скобках после
- имени класса, при исп класса заменяется типом
- public class myClass<T>{
- private T a;
- public holder(T a){this.a=a}
- public T get(){return a;}
- }
- Main -> Holder<Cat> = new Holder<Cat>(myCat) -< не можем добавить не кошку
- Можно размещать только объект указанного типа
- Принцип обобщения - указываете, какой тип использовать и используете
- В общем случае работа с обобщениями как и с другим типом
- Обобщение можно использовать указанием имен со списком аргументов-типа

- Библиотека кортежей
- При вызове метода возникает необходимость вернуть несколько объектов
- return - только один -> создание объекта, содержащего несколько объектов (можно через класс, но
- можно по другому)
- Группа объектов - завернутых в другой объект - кортеж
- Получатель может читать элементы, но не добавлять новые
- Кортеж имеет произвольную длинну
- public class TwoTuple<A,B>{
- public final A first;
- public final B second;
- public TwoTuple(A a, B b)}
- - конструктор сохраняет объекты в кортеже
- путем финализации - получаем только гетеры к полям

- Для создание кортежей большей длинны - наследование - наследник <A,B,C> - в конструкторе
- super(a,b)
- third = c; <- final поле наследника

- Чтобы использовать кортеж - определяете объект с нужной длинной как возвр. знач. функции

- Класс стека - реализация стека LinkedList, но можно собсвенный механизм, основанный на LinkedList
- Public LinkedStack<T>{
- private Node<T> top = new Node<T>();
- void push (T item){
- top = new Node<T>(item,top);
- }
- public T pop(){
- T res = top.item
- if(!top.end())
- top=top.next;
- return res}
- }
- внутри класс
- private static Node<U>{
- U item;
- Node<U> next;
- Node(){item=null;  next = null;}
- Node(U item, Node<U> next){
- this.item = item;
- this.next = next;}
- bollean end() {return item==null&& next == null}}

- При каждом методе push создается новый узел Node и связывается с предыдущим

- RandomList - тоже может быть класс с параметризированным типом на основе ArrayList, с включением
- функции select и выдачи рандомных значений

- Обобщенные интерфейсы
- Обобщения работают с интерфейсами
- Генератор - класс для создания объектов (похож на фабричный метод, без аргументов)
- В генераторе метод next - создающий новые объекты + параметризация метода по <T>

- Обощенные методы
- Класс, содержащий обощенные методы не обязательно обобщенный
- Обобщенный метод изменяется не зависимости от класса, лучше обобщать методы чем классы
- public<T> void f(T x){
- System.out.println(x.getClass)}

- В обобщенных классах параметры-типы должны указываться при создании экземпляра. Для методов это необязательно
- компилятор может вычислить их за вас -> f() как обычные вызовы методов. Для вызовов с примитивными типами
- идет автоматическая упаковка преобр типа в объекты

- Автоматическое определение аргументов-типа
- Обобщение -> расширение кода
- Механизм автоматического определения аргументов-типов способен приветси к упрощению. Можно создать
- класс с различными статич методами с наиболее часто-исп типами и вызывать оттуда;
- Итог - устранение необходимоти в повторении списка парамтеров
- Автоматическое определение работает только при присваивании!

- Явное указание типа
- Тип можно задать явно тип указывается в угловых скобках, после точки, перед именем метода
- При вызове метода перед точкой - this, для статических методов - имя класса перед точкой

- Списки аргументов переменной длинны
- Обобщенные методы нормально существуют со списками аргументов переменной длинны
- пример -
- public static <T> List<T> makeList(T... args){
- List<T> result = new ArrayList<T>();
- }

- Обобщенный метод для использования с генераторами/итераторами
- заполнения коллекций с обобщенным методом -
- fill(Collection<T> coll, Iterator<T> gen, int n) - > внутри заполняем следующими
- Создание генератора - class BasicGenerator<T>
- -private Class<T> type;
- public BasicGenerator(Class<T> type){this.type=type;}
- public T next(){ return type.newInstance}
- public static <T> Generator<T> create(Class<T> type){
- return new BasicGenerator<T>(type);}
- Базовая реализация для создания объекта октрытого класса с конструктором по умолчанию

- Упрощение использование кортежей.
- Оформление кортежей в виде библиотеки общего назначения
- Для создание перегруженный метод
- -Класс без парамтеров, внутри класса функции с параметрами по созданию объектов с параметрами представляющими
- из себя кортежи разной длинны
- - один из параметров
- public static <A,B,C> ThreeTuple<A,B,C> tuple(A a, B b, C c){return new ThreeTuple(a,b,c)}

- Операции с множествами
- использование множеств Set - удобно определять в виде обобщенных методов, которые могут исп.
- с любыми типами
- Объединение -
- public static <T> Set<T> union(Set<T> a, Set<T> b){
- Set<T> result = new HashSet<T>(a);
- result.addAll(b);
- ret res}
- Пересечение
- ... intersection(Set<T> a, Set<T> b){
- Set<T> result = new HashSet<T>(a);
- return res.retrainAll(b)}
- Разница
- public static <T> Set<T> difference(Set<T> super, Set<T> SUB){
- res = hashset(super);
- return res.remobeall(SUB);}

- Анонимные внутренние классы
- Также могут использоваться с обобщениями.

- Преимущество обобщенй - возможность простого сздания сложных моделей. - можно создать лист с кортежами

- Згадка стирания
- запись ArrayList<T>.class - недопустима
- ArrayList<Integer>.getClass==ArrayList<String>.getClass
- Class.getTypeParametrs - массив объектов тип, представляющий объект переменных типа - возвращает
- только индефикаторы, представляющие параметры (для карты - k v)
- В обобщенном коде информация о праметрах-типах обобщения недоступна

- Обобщения реализуются с помощью стирания - любая информация о типе теряется, известно, что вы используйте объект
- поэтому два класа до этого равны

- Чтобы вызвать определенную функцию у параметра необходимо передать ему ограничение
- <T extends *Класс содерж функцию*>
- Тогда параметр стирается до первого ограничения
- Но в таком случае можно вообще не проводить обобщения
- -> обобщения используются когда вы хотите использовать более универсальные параметры-типы
- T extends может быть полезен когда класс сродержит метод, возвращающий T
- Миграционная совместимость
- Обобщенный тип только во время проверки типов, после чего он заменяется необобщенным
- верхним ограничителем -> List<T> стирается до List
- Позволяет использовать код с необобщ библиотеками - Миграционная зависимость
- Обобщения должны обеспечивать обр совместимость + миграц совместимость
- Каждая библиотека и приложение не должны зависеть в отношении исп обобщений -> все признаки
- обобщений должны быть стерты
- Процесс перехода от необобщ кода к обобщ и возможность встраивания библиотек -> основные
- причины использования стирания
- Но обобщенные типв не могут быть задействованы в операциях, где явно задействуются типы (new,
- instanceof)
- MyClass<Cat> myClass = new MyClass<Cat>();
- но внутри класс не знает, что он работает с кошкой

- Для отключения предупреждений существуют аннотации
- @SuppressWarnings("unchecked") -> включается в метод генерирующий предупреждения

- Граничные ситуации
- private Class<T> kind;
- (T[])Array.newInstanse(kind, size) -> kind хранится в виде Class<T> -> механизм стирания ->
- хранится только класс без параметра -> масив nulleй

- Любые операции, требующие знания точного типа не будут работать с T
- Стирание иногда приходится компенсировать используя меки типа.
- Явная пеедача объекта Class для своего типа, чтобы его можно было использовать в выражениях типов
- -> new MyClass<SomeType>(SomeType.class);

- Создание экземпляров типов
- Попытка создания new T() - неудача, тк стирание + отстутсвие констрк по умолчанию
- Создание new T можно релизовать с объектом-фабрикой
- В качестве фабрики удобно использовать объект Class, при использовании метки можно воспользоваться
- newInstance()
- class ClassAsFactory<T>{
- T x;
- public ClassAsFactory(Class<T> kind){
- try{
- x = kind.newInstance()}
- catch(Exception e){
- throw new RuntimeException(e);}}}
- Но если попытаться создать ClassAsFactory<Integer> - ошибка, тк Integer не имеет конструктора по
- умолчанию
- Вместо этого рекомендуется использовать явную фабрику и добавлять класс, в котором используется эта фабрика
- interface Factory<T>{
- T create()} -> наследуется и реализуется для каждого класса, можно использовать с параметризованными
- классами
- class Foo2<T>{
- private T x;
- public <F extends Factory<T>> Foo2(F factory){
- x = factory.create();}}
- Можно использовать фабричный метод вместо фабрики
- abstract class GenericWithCreate<T>{
- final T element;
- abstract T create();
- GenericWithCreate(){element=create();}}
- class x {}
- class Creator extends GenericWithCreate<X>{
- X create(){return new X();}
- void f(){
- System.out.println(element.getClass().getSimpleName());
- }}
- public class Gereric{
- Creator c = new Creator;
- c.f} -> создаст x;

- Массивы обобщений
- Создать массив обобщений невозможно. -> использование ArrayList
- Созание массива обобщенного типа - создание массива стертого типа с последующим привидением
- public class GenericArray<T>{}
- private T[] array;
- public GenericArray(int sz){
- array = (T[])new Object[sz];}
- public void put(int index;T item){
- aray[index]=item;}
- public T get(int index) {return array[index];}
- public T[] rep(){return array} - возвращает массив надлежащего вида
- можно пользоваться только через new, при обращении к rep - исключение (если не Object)
- Использование T[] array = new T[sz] - нельзя
- Из-за стирания тип массива - object
- В коллекции лучше использовать Object[], а выполнять приведение к T при использовании элемента
- массива., тогда rep - сработает

- Ограничения
- Сужают диапазон параметров-типов, используется extends, можно несколько классов, используя &
- Можно вызывать функции, принадлежащие классам

- Маски - вопросительные знаки в выраж обобщ арг
- Массив производного типа можно присвоить ссылке на массив базового
- Fruit[] = new Apple() -> в массив помещаются яблоки и производные от яблок
- При момещении просто фруктов или фруктов не яблок - ок, но во время выполнения не ок
- Если сделать подобное в параметре - <> -то вызовет ошибку -> обобщение с яблоками не есть обобщение
- с фруктами. Нетпроблем в компиляции потому что идет сравнение типа контейнера
- Для установления разновидности восходящ отнош используются маски
- List<? extendce Fruit> - можно хранить некоторый конкретный тип фрукта(но не все сразу)
- - такой объект может указывать на List<FruitType>
- - такой объект может хранить фрукты, приведенные к типу фрукт

- При обращении к контейнеру с параметризацией с функциями contains и indexOff, то аргументы- объекты
- маски не задействованы, компилятор разрешает вызов
- При обращении с функцией add - аргумент - ? extendce Fruit - непонятно, что именно, не принимает
- ничего
- Можно реализовать гет\сет и в качестве параметра тип - в случае с ? extendce Fruit - максимальный
- возвращаемый парметр фрукт. Если положить яблоко и попытаться достать апельсин из фруктов
- через скобки -не получится
- метод сет тоже не будет работать с маской потому что не знает, какой тип
- работает только equals() который через обджект

- Контравариантность
- Ограничение супертипа - органичение по любому базовому классу некоторого класса - <? super MyClass>
- - безопасная передача объекта к обращенному типу
- <? super Apple> - можно передать яблоки и все ниже яблок, фрукты нельзя
- Ограничение супер типа ослабляют то, что можно передать в контейнер
- writeIt(List<? super T> list, T item)
- {
- list.add(item);
- }
- List<Apple> = new ArrayList<Apple>();
- List<Fruit> fruit = new ArrayList<Fruit>();
- writeIt(apples, new Apple())
- writeIt(fruit, new Aplple())
- В случае статического метода можно использовать и List<T>, тк адаптируется к каждому вызову

- Неограниченные маски
- Маска <?> - что угодно - эквивалентно использованию самого типа
- Map<?,?> = Map
- Но <?> и Object рассматриваются по разному - Object - object, ? - конкретный тип, который мы не знаем

- Неспециализированный тип -> при гет - может быть возвр только object
- set -> передача любого объекта который по итогу буде преобр к Object;
- При использовании не спец типа, отказываетесь от проверки на стадии компиляции
- Но не спец Holder - комбинация любых типов
- Holder<?> - один конкретный тип
- если передать неспец ссылку Holder методу без маски (получает точный тип) -> предупреждение
- при испоьзовании <? extendce T> - ограничение на set, чтоб не добавить не то, что нужно, но без ограничений
- на get

- С маской супер типов - сет работает для всего, что работает с бащрй
- гет не очень, тк может быть любой супер тип внутри
- Если аргумент должен получать Holder<type>, то имеет больше всего ограничений

- Преимущество использование точных типов - большая функциональноть с обобщ парамерами
- Использование масок - более широкий диапазон принимаемых типов

- Фиксация - использование <?> - необходимо, при передачи не спец тип, автоматически вычесляет параметр-тип
- и вызывает другой метод - механизм фиксации
- static <T> void f1(Holder<T> holder){
- T t = holder.get();
- }
- static void f2(Holder<?> holder){
- f1(holder); // - вызов с зафиксированным типом
- }

- Работает для вызова но не для записи

- Проблемы
- Примитивы не могут быть параметрами-типами (List<int> - нельзя), обертки - можно
- ! Автоматическая упаковка не применяется к массивам

- Реализация параметризованных интерфейсов
- Класс не может реализовывать две разновидности одного интерфейса
- Из-за стирания - один интерфейс
- interface Payable<T>{}
- class Employee implements Payable<Employee> {}
- Class Hourly extends Employee implements Payable<Hourly>
- нельзя именно из-за наследование интерфейса

- Использование приведения и instanceof -> ничего
- Можно хранить значения как обджект (через массив) и преобразовывать для выдачи,
- используя SuppressWarnings("unchecked")
- Приведение можно осуществить к общему классу, но не к типу
- in = ObjectInputStream
- List.class.cast(in.readObject())

- Перегрузка
- void f(List<T> v){}
- void f(List<W> v){}
- нельзя
- необходимо указать имена различные имена методов (потому что листы могут бытьодинаковыми)

- Перехват интерфейса базовым классом

- имеется класс Pet реализующий сравнения с другими объектами Pet
- ... implements Comparable<Pet>
- Классы наследники не могут сужать тип для сравнения, тк родитель наслдедует сравнение именно со своим
- типом

- Самоограничиваемые типы
- class SelfBounded<T extends SelfBounded<T>>{//...}
- Версия идиомы в которой нет самоогран -
- class Type<T>{}
- class RecurringType extendce Type<ReccuringType>
- - создание базового класса, использующего производный тип в аргументах и значениях
- Базовый класс заменяет свои параметры производным
- Самоограничение
- Заставляет обобщение использовать самого себя в качестве ограничения
- параметр тип совпадает с определяемым классом
- Принудительное соблюдение отношений наследования
- Ковариантность агрументов
- Ковариантные типы аргументов - типы аргументов методов меняются в соответствии с субклассами
- В ограниченном типе для субклассов возможно использование только субклассов -> для сет нельзя поместить базовый тип
- Без самоограничения - можно

- Динамическая безопасность типов
- Контролируемый контейнер - checkeКонтейнер - не допускает помещение объектов другого типа
- не контролируемый - допускает, но выдает ошибку при попытке получения объекта

- Исключения-
- блок catch не может перехватывать исключения обобщенного типа, тк тип искл должен быть известен во
- время компиляции и во время выполнения
- Обобщенный класс не может наследовать от Throwable - препядствует определению обобщенных исключений
- Но параметры-типы могут исп в секции throws объявления метода

- Примеси
- Смешение функциональности нескольких классов для получение итогового класса.
- Если вы измените что-то в примеси, изменения распр на все классы, к которым примесь применяется
- Для достижение эффекта примесей рекомендуется использовать интерфейсы
- Похоже на делегирование

- Паттерн Декоратор
- концепция примеси похожа на декоратор
- Используются в ситуациях, когда для обеспечения всех возможных комбинаций субклассирование ->
- слишком много классов
- Декоратор - иерархия объектов для добавления обязанностей в объекты
- Все обертки - обладают единым интерфейсом
- Имеется объект, функциональность которого дополняется за счет оберток
- -> имеется общий набор сообщений, отправляемых объекту
- Декараторы - структуры Примеси - наследование
- Объект, получаемый с помощью декоратора - последний, с которым работали

- Примеси и динамич заместители (прокси)
- создают механизм, точнее моделирующий примеси
- Динамически тип полученного класса -> соединение всех смешиваемых

- Латентная типизация
- Для написания широко применяющегося кода необходимы механизмы ослабления ограничений на типы,
- с которыми он может рабоать
- Обобщение - шаг к этому
- Проблема - работа с обобщенными типами
- В некоторых ЯП проблема решается латентной типизацией -> отказ от конкретного типа, фокусировка
- на методах, которые можно вызвать питон\плюсы
- Можно реализовывать интерфейсы и обязывать классы делать что-то
- Компенсация через
- Отражение
- class Refl{
- static void perfom(object){
- Class<?> spkr = obj.getClass
- try Method speak = spkr.getMethod("speak")
- try Method sit = spkr.getMethod("sit")}}

- Применение метода к последовательности
- Отражение передает всю проверку на стадию выполнения
- Нужно реализовать apply(), применяющий произв метод к каждом объекту последовательности
- Apply должен реализовывать Iterable + может принять любой объект реализовывающий Iterable

- Если нет нужного интерфейса
- Метод должен наследовать функцию добавление -> берем и требуем коллекцию, каждая поддерживает
- add

- Адаптеры для латентной типизации
- Латентная типизация задает неявный интерфейс, содержащий нужные методы
- Написание кода для нужного интерфейса по имеющемуся - пример Адаптера

- Использование объектов функций как статегий

- Задача - написать функцию сложения для разных типов
- Сложность - разная реализация + для разных типов
- Решение - использование паттерна Стратегия -> изолирует то, что изменяентся, внутри объекта
- функции. Объект функции - объект, который в какой-то мере ведет себя как функция, содержит
- один метод. В отличие от обычных методов могут передаваться при вызовах + иметь сост. поддерживаемое
- между вызовами (как поле - функция).
- Контекст - настаивается конкретной стратегией
- хранит ссылку на объект класса Strategy
- Может определять интерфейс,который позволяет объекту Strategy обращаться к данным контекста
- Конкретная стратегия - реализует алгоритм, объявленный в классе Стратегия
- Стратегия - объявляет общий для всех алгоритмов интерфейс. Класс Контекст пользуется этим интерфейсом
- для опеределения конкретного алгоритма
- Функции (конкретные стратегии) в виде
- static class ConcreteStrategy impelements Strategy<Type>{
- public Type func(Type x){
- funcforx(x);}}

# Самый лучший механизм для использования обобщенных типов - классы контейнеры .