1 Параметризация и обобщенное программирование

Параметризация позволяет использовать тип как параметр при определении класса, интерфейса или метода. Это позволяет переиспользовать один код для различных типов входных параметров.

Также параметризированный код дает дополнительные преимущества:

- Проверка типов на стадии компиляции. Лучше исправлять compile-time error, чем runtime error.
- Нет необходимости вручную приводить типы.

```
// without generics
List list = new ArrayList();
list.add("Hello");
String s = (String) list.get(0);

// generics
List<String> list = new ArrayList<>();
list.add("Use generics everywhere!");
String s = list.get(0);
```

- Обобщенные алгоритмы. Java Collections невозможно представить без параметризации - контейнеры могут хранить любой тип данных.

Объявление

Компилятору можно принудительно указать, с каким типом данных работает коллекция. В качестве параметра нельзя использовать примитивные типы данных.

```
List<String> strings = new ArrayList<String>();
List<String> strings = new ArrayList<>();
StringBuilder builder = new StringBuilder();
for (String s : strings) {
   builder.append(s);
}

Map<Manager, List<Employee>> staff = new HashMap<>();
// Look through the map
for (Map.Entry<Manager, List<Employee>> entry : staff.entrySet()) {
   Manager manager = entry.getKey();
   List<Employee> employees = entry.getValue();
}
```

Начиная с версии Java 7 можно использовать diamond operator <> в правой части выражения, и компилятор самостоятельно выведет правильный тип.

Чтобы определить параметризованный класс используется синтаксис Some<T1, T2, T3,..> , где Ti - это тип. По стандартному соглашению в качестве имен параметров используются буквы, передающие смысл параметра:

- E - Element
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

В коде класса вместо конкретного типа используется параметр.

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
   private T t;

   public void set(T t) { this.t = t; }
   public T get() { return t; }
}

/**
 * @param <K> the type of key
 * @param <V> the type of value
 */
public interface Pair<K, V> {
   K getKey();
   V getValue();
}
```

Чтобы объявить параметризованный метод, в его сигнатуре следует указать используемые типы перед типом возвращаемого значения.

```
public class Util {
   // Generic static method
   public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
```

```
        return p1.getKey().equals(p2.getKey()) &&
             p1.getValue().equals(p2.getValue());
    }
}
```

Массивы и параметризация

Нельзя инициализировать параметризованный массив. Во время исполнения программы вся информация о типах стирается (type erasure) и компилятор работает с типом Object. То есть информация о типах существует только на момент компиляции. В то же время, массивы должны знать о типах данных во время исполнения (Нужно знать тип элемента при создании массива). Объявлять параметризованные массивы разрешено.

```
E[] elements = new E[size]; // compile-time error

// But you can declare an array
//
Objects[] objects = new Objects[size];
@SupressWarning("unchecked")
E e = (E) objects[0];

// or this way
E[] elements; // ok
@SupressWarning("unchecked")
elements = (E[]) new Objects[size];
E e = elements[0];
```

Ограничения и маски

При использовании параметризации компилятор проверяет точное совпадение типов. Однако иногда нужно расширить функционал и позволить обрабатывать объекты дочерних или родительских классов.

T extends Some - элементы класса Some и всех его потомков.

T super Some - элементы класса Some и всех его родителей.

```
// Number and all sub-types are allowed
public static <T extends Number> T getFirst(List<T> numbers) {
    T t = numbers.get(0);
    return t;
}
```

```
// Integer and all supertypes are allowed
pubic static <T super Integer> void fillContainer(List<T> list) {
   Integer i = 10;
   list.add(i);
}

public static void test() {
   // class Integer is a subtype of Number
   List<Integer> integers = Arrays.asList{1, 2, 3, 4};
   Integer first = getFirst(integers);

   // Number is a supertype of Integer
   List<Number> numbers = new ArrayList<>();
   fillContainer(numbers);
}
```

Wildcards

Если нам не важно, какой тип используется и мы не возвращаем из метода параметризованное значение, то можно указать просто <?>. В таком случае мы не будем иметь никакой информации о типе, кроме того, что это Object.

```
   public static void printAll(List<?> list) {
      for (Object o : list) {
         System.out.println(o);
      }
   }
```

Wildcards могут использоваться вместе с ограничениями super, extends. Рассмотрим пример структуры данных стек и две операции на ней - извлечь все элементы и поместить все элементы в стек.

```
public class GenericStack<E> {
   // ...

   // push all elements from @src to stack
   public void pushAll(Collection<? extends E> src) {
      for (E e : src) {
         push(e);
      }
   }

   // pop all elements from stack to @dst
```

```
    public void popAll(Collection<? super E> dst) {
        while (!isEmpty()) {
            dst.add(pop());
        }
    }
}
```

В методе pushAll(Collection<? extends E> src) коллекцию src можно рассматривать как поставщика данных (producer). Поставщик может предоставлять элементы дочерних типов E.

В методе popAll(Collection<? super E> dst) коллекция dst - потребитель данных (consumer). Потребитель может принимать значения супертипов E.

PECS

PECS - Producer Extends, Consumer Super - мнемоническое правило для написания правильных параметризованных методов для обработки коллекций.

Producer Extends - если нужна коллекция, которая поставляет данные (то есть, из нее читают), объявите ее <? extends T>

ConsumerSuper - если нужна коллекция, которая потребляет данные (то есть, в нее пишут), объявите ее <? super T>

Наследование

```
List<String> strList = new ArrayList<String>();
List<Object> objList = strList; // Compile time error
List rawList = strList; // OK - using raw type
```

Несмотря на то, что String является наследником Object, List<String> не является наследником List<Object>.

Иерархия для ArrayList<String> -> List<String> -> Collection<String> -> Object

```
List<String> strList = new ArrayList<String>();
Collection<String> col = strList;
Object obj = col;
```

Задачи

1. Дан интерфейс Stack.

```
public interface Stack<E> {
   // add new element to the top of the stack
   public void push(E element) throws StackException;
   // return and remove an element from the top
   public E pop() throws StackException;
   // return the top element but doesn't remove
   public E peek();
   public int getSize();
   public boolean isEmpty();
   public boolean isFull();
   // add all elements from @src to the stack
   public void pushAll(Collection<? extends E> src) throws
      StackException;
   // pop all elements from stack to @dst
   public void popAll(Collection<? super E> dst) throws
      StackException;
}
```

   Реализуйте класс class GenericStack<E> implements Stack<E>. Также реализуйте класс StackException extends Exception - исключение должно кидаться, если стек полон и кто-то вызывает метод push() или если стек пуст и кто-то вызывает pop().

2. Есть коллекция элементов String. Вам нужно сортировать эту коллекцию по длине строки. Используйте Collections.sort(List<T> list, Comparator<? super T> cmp).

```
// You need to implement LengthComparator implements Comparator
public static void test() {
   List<String> s = Collections.asList{"aaa", "b", "cd"};
   // Should return {b, cd , aaa}
   Collections.sort(s, new LengthComparator());
}
```

3. (*) Напишите тесты к своим решениям, использую библиотеку JUnit

4. (*) Для вывода отладочной информации используйте систему логирования

5. Финансовый менеджер. Приложение для контроля своих расходов. Каждый пользователь имеет логин и пароль. У пользователя могут быть несколько счетов. Каждый счет имеет текстовое описание, текущий остаток и список транзакций.

У транзакции есть метка пополнение-снятие, дата проведения, сумма и описание. Нужно реализовать классы User, Account, Record и интерфейс DataStore.

Имя пользователя должно быть уникальным, каждый счет и транзакция также имеют уникальный номер. Вы должны уметь сравнивать два объекта (посмотрите, что такое equals(), hashCode()).

(*) Напишите comparator, который сортирует список транзакций по дате

(*) Напишите тесты к своей реализации

(*) Напишите процедуру авторизации и регистрации. Это может быть графический или консольный интерфейс, нужно дать пользователю возможность зарегистрироваться (проверив, что имя уникальное) и зайти в приложение (проверив правильность пароля).

```
public interface DataStore {
   // return null if no such user
   User getUser(String name);
   // If no users, return empty collection (not null)
   Set<String> getUserNames();
   // If no accounts, return empty collection (not null)
   Set<Account> getAccounts(User owner);
   // If no records, return empty collection (not null)
   Set<Record> getRecords(Account account);
   void addUser(User user);
   void addAccount(User user, Account account);
   void addRecord(Account account, Record record);
   // return removed User or null if no such user
   User removeUser(String name);
   // return null if no such account
   Account removeAccount(User owner, Account account);
   // return null if no such record
   Record removeRecord(Account from, Record record);
}
```

Логирование и тесты

Для тестирования используем JUint4 (junit.org) Для логирования Logback, SL4J (http://logback.qos.ch/ , http://www.slf4j.org/) Все необходимые библиотеки доступны на сайтах выше, а также в архиве к этому уроку. Вам достаточно добавить их к проекту (способ зависит от среды разработки)

Чтобы включить логирование объявляем в коде

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Some{
      static Logger logger = LoggerFactory.getLogger(Some.class);

      void test() {
            logger.warn("Warn message");
      }
}
```