Лекция 4

Загадка

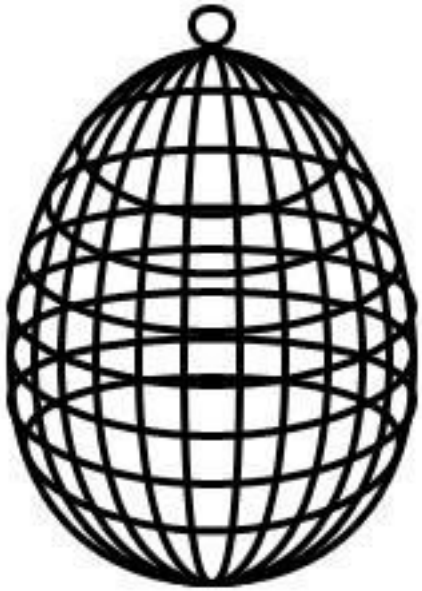
Бейсбольная бита и мяч вместе стоят $1.10 (доллар и 10 центов). Бита ровно на доллар дороже, чем мяч. Сколько стоит мяч (в центах)?

Загадка

В озере есть участок, заросший кувшинками. Каждый день этот участок увеличивается в размере в 2 раза. Известно, что кувшинки покроют всю поверхность озера за 48 дней.
За сколько дней кувшинки покроют ровно половину

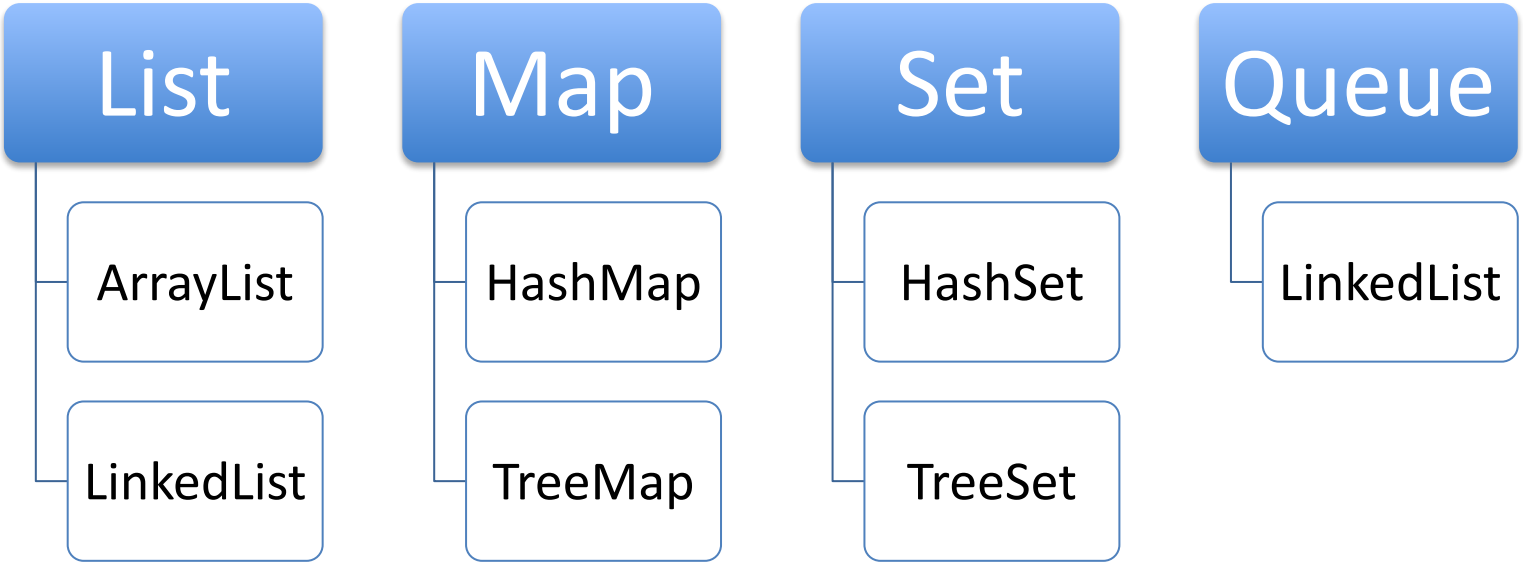
Загадка

Загадка



Коллекции

1. **Коллекции – это хранилища объектов.**
2. **С ними работать очень удобно когда у вас множество одинаковых объектов**
3. **В Java коллекции описаны следующими интерфейсами:**



List

1. List – это динамический список(массив). Это не класс, а интерфейс.
2. Интерфейс List описывает как должен работать динамический список. В нем есть следующие методы:
   add() – добавление нового элемента
   remove() – удаление элемента
   size() – размер списка
   contains() – проверяет содержит ли список указанный элемент.
3. При объявлении списка, можно и нужно указать тип хранимых объектов в списке.

```
List<Type> list; // вместо Type пишется тип
```

4. Например, список строк:

```
List<String> strs;
```

5. Или список целых чисел

```
List<Integer> ints;
```

Все коллекции могут работать только с классами. Вместо примитивных типов, нужно использовать их обертки:
int – Integer, double – Double, char – Character, boolean – Boolean

List

1. ArrayList - это класс, который реализует интерфейс List.
   Так создается список строк:
   ```
   List<String> names = new ArrayList<String>();
   ```
2. Добавление новых элементов:
   ```
   names.add("Elizabeth");
   String kingsName = "Arthur";
   names.add(kingsName);
   ```
3. Получение количества элементов:
   ```
   int count = names.size(); //вернет 2
   ```
4. Удаление какого-либо элемента:
   ```
   names.remove("Elizabeth");
   ```
5. Очищение списка:
   ```
   names.clear();
   ```
6. Перед добавлением или удалением, вы можете проверить есть ли этот элемент в списке:
   ```
   if (names.contains("James")){
         names.remove("James");
   }
   ```

List

2. Списки можно перебирать циклом for:

```
for(String name: names){
        System.out.println(name);
}
```

3. Списки могут быть любого типа:

```
List<Car> cars = new ArrayList<Car>(); //список машин
List<Printer> printers = new ArrayList<Printer>;
```

4. Элементы можно получать по индексу как в массивах:

```
printers.get(0); //получить 0-вой элемент

names.get(12); //получить 12-й элемент
```

5. Если элемента с таким индексом нет, вы получите IndexOutOfBoundsException

Set

1. Set – это неупорядоченное множество уникальных элементов. В отличие от List, в Set нельзя добавлять повторно один и тот же элемент.
2. Set содержит такие же методы как у List:
   add(), remove(), size(), contains()
   кроме get() потому что из множества нельзя получить определенный элемент. Его элементы можно только перебрать циклом.
3. Или проверить содержит ли множество какое-либо значение.
   contains()
4. Так же при объявлении желательно указывать тип хранимых элементов:
   Set<String> strs; // множество строк
   Set<Double> doubles; //множетсо Double

Set

1. Стандартная реализация интерфейса Set это HashSet:

   Set<String> books = new HashSet<String>()
2. Добавление

   books.add(“Harry Potter 1”);

   books.add(“Master and Margaret”);

3. Удаление :

   books.remove(“Harry Potter 1”);

4. провека существования элемента во множестве:

   books.contains(“Sherlock”);

5. Даже если добавить один и тот же элемент два раза, множество будет содержать этот элемент всего один раз, в отличие от списка.

Map

1. Map – словарь или ассоциативный массив - это множество пар, где каждому ключу соответствует определенное значение.
2. В словаре можно хранить разные значения по определенному уникальному ключу. И каждый уникальный ключ будет соответствовать определенному значению

   “first name” - > “Jonathan”
   “last name” -> “Livingston”
   “city” -> “New York”

3. Так как и словаря есть ключ и значение, для него нужно указывать сразу два типа при объявлении: тип ключа и тип значения:

   Map<String, String> info;

Map

1. HashMap – стандартная реализация интерфейса Map.
   Map<String, Integer> scores = new HashMap<String, Integer>;

1. Добавление нового значения по ключу:
   scores.put("Marina", 3); // У Марины три очка
   scores.put("Syrym", 4); // У Сырым 4 очка
2. Удаление ключа со значением:
   scores.remove("Syrym");
3. Получить значение можно по ключу:
   scores.get("Marina"); // получим 3
4. Проверить содержит ли словарь, данный ключ:
   scores.containsKey()
5. Проверить содержит ли словарь, данное значение:
   scores.containsValue()
6. Получить все ключи:
   Set keys = scores.keySet();
7. Получить все значения:
   Collection values = scores.values();

Map

1. **Пример: Допустим вы хотите хранить разную информацию о каком либо человеке и заранее не знаете сколько информации может быть о нем. Для этого вы можете создать словарь и добавлять в него любую информацию по мере нахождения:**

```
Map<String, String> info = new HashMap<>();

info.put("Имя", "Рапунцель");
info.put("Должность", "Принцесса");
info.put("Особенность", "Волшебные волосы");
info.put("Родина", "Арендел");
```

1. **Потом вы можете все это обработать или сохранить или вывести, перебрав все элементы:**

```
for(String key: info.keySet()){
   String value = info.get(key);
   System.out.println(key+": "+value);
}
```

```
Должность: Принцесса
Родина: Арендел
Особенность: Волшебные волосы
Имя: Рапунцель
```

Queue

1. Queue – это очередь. Т.е. Коллекция которая работает по принципу FIFO – “Первым вошел – первым вышел”. Т.е. В очередь можно добавлять элементы в любом порядке. А получать элементы из нее только по одному в порядке добавления.

1. У очереди есть методы:
   add() – добавление элемента в конец очереди
   poll() – получить первый элемент и удалить его с очереди
   peek() – получить первый элемент, но не удалять его
   remove() – просто удалить первый элемент
   isEmpty() – проверка пустоты очереди
1. Очередь используется когда важен порядок элементов. Например: Диспетчер задач может иметь список задач на выполнение. Задачи поступают в разное время, и диспетчер должен выполнить их в порядке поступления.
2. Или очередь покупателей. Какой покупатель пришел

Queue

1. **LinkedList – это стандартная реализация очереди.**
   Queue<Задача> очередь = new LinkedList<Задача>();
2. **Как примерно выглядит добавление в очередь и ее обработка:**

```
Queue<Задача> очередь = new LinkedList<Задача>();

очередь.add(new Задача("Какая то задача 1"));
Задача задача2 = new Задача("задача 2");
очередь.add(задача2); // один и тот же элемент
очередь.add(задача2); // можно несколько раз добавлять

while (!очередь.isEmpty()){
   Задача текущаяЗадача = очередь.poll();
   текущаяЗадача.запустить();
}
```

Generics

1. Все эти описанные коллекции являются «Обобщенными» типами или “Generic types”.
2. Generic types или Generics – это типы которые могут иметь параметры.
3. И как параметрами для них выступают другие типы данных.
4. Параметры указываются внутри треугольных скобок рядом с именем типа:
   List<String>, ArrayList<Car>, Map<String, Integer>, Queue<Задача>

1. Такие типы должны уметь работать с любыми другими типами, т.е. быть универсальными.
2. Вы также можете создавать Generic типы или классы
3. Для этого при описании класса или интерфейса, должны указать что ваш класс принимает параметр внтури треугольных скобок:
   public class Box <T>

Generics

1. Реализуем класс коробку, в которую можно положить любой другой объект, но только один. При этом вначале указав какого типа объект может в себе держать коробка:

```
public class Box<T> {    //T - это параметр-тип, он может быть любым
    private T element;   //Поэтому мы заранее не можем сказать
                         // какого типа будет element

    public T getElement() {
        return element;
    }

    public void putElement(T element) {
        this.element = element;
    }
}
```

1. Создание объекта коробки, которая может держать машину в себе:

```
Box<Car> carBox = new Box<Car>(); //Car будет подставлен вместо T
Car car = new Car();
carBox.putElement(car);
```

Обработка ошибок

1. Программа не всегда может работать так, как ожидается от нее.
2. На это могут повлиять внешние факторы как: Память, ресурсы, пользовательское вмешательство и т.п.
3. На это могут повлиять случаи, которые не учел программист. Т.е. Исключительные ситуации.
4. И когда ваша программа не знает что дальше делать, она «вылетает» или «кидает» ошибку.
5. Когда программа вылетает, в обычно видите информацию о том в каком месте произошла ошибка. Это **StackTrace**

```
Exception in thread "main" java.util.NoSuchElementException    at
java.util.LinkedList.getFirst(LinkedList.java:242)    at
java.util.LinkedList.element(LinkedList.java:661)    at
collections.Main.main(Main.java:33)
```

6. Чтобы пользователь не чувствовал дискомфорта при использовании нашей программы или чтобы программа не вылетала просто, мы должны уметь

Обработка ошибок

1. StackTrace показывает какая последовательность вызовов методов была произведена перед тем как произошла ошибка. И читается снизу вверх. Т.е. Нижные вызовы были первее верхних.

```
Exception in thread "main" java.util.NoSuchElementException    at
java.util.LinkedList.getFirst(LinkedList.java:242)    at
java.util.LinkedList.element(LinkedList.java:661)    at
collections.Main.main(Main.java:33)
```

1. StackTrace помогает нам понять причину ошибки и обнаружить его местонахождение, показывая тип ошибки, в каком файле и в какой строчке произошла ошибка.
2. Если мы знаем в каком месте произойдет ошибка, мы можем ее обработать, не останавливая выполнение программы. Это делается с помощью операторов try, catch

```
try {
    очередь.element(); //здесь может произойти ошибка, если очередь
пустая
} catch(NoSuchElementException e){
    ... //сделать что нибудь или проигнорировать
}
```

Обработка ошибок

1. **Внутри оператора try { } может находиться любой код, который может вызвать исключительную ситуацию.**
2. **После try всегда идет один или несколько операторов catch, каждый из которых ловит определенный тип ошибок или исключительных ситуаций. Поэтому в операторе catch указывается тип ошибки. Внутри оператора catch может быть любой код, который как то обработает пойманную ошибку. Пример:**

```
try{
    очередь.element();
}
catch(NoSuchElementException e){ //Элементов нет
    System.out.print("Очередь пустая");
}
catch(NullPointerException e){ // Очередь не инициализирована
    System.out.print("Очередь еще не создана");
}
```

Обработка ошибок

1. Ошибки – это обычные классы, которые наследуются от классов Error, Exception.
2. По умолчанию наследники от Error – это серьезные системные ошибки, связанные с ОС или Java машиной, которые нельзя обработать в основном никак. Например: VirtualMachineError, ThreadDeath
3. Наследники Exception – это не очень серьезные ошибки, которые не причиняют большого вреда программе, и их можно и нужно обработать. Например: EOFException, NullPointerException, NoSuchElementException.
4. Java не позволит вам скомпилировать программу, пока вы не обработаете все необходимые исключительные ситуации.
5. Есть еще класс RuntimeException, наследников которого Java позволяет не обрабатывать, но соответственно они не должны причинять никакого значимого вреда программе.

Обработка ошибок

1. Можно так же создавать свои классы ошибок и исключительных ситуаций.
2. Для этого нужно унаследовать от какого либо класса ошибок. Например:

```
public class CartridgeException extends Exception {
   …
}
```

1. Чтобы вызвать ошибку в коде, используется оператор ***throw***
2. ***Например вызовем ошибку, если краска закончилась в принтере:***

```
public void print(String text) throws CartridgeException {
   if(cartrdge.isEmpty()){
      throw new CartridgeException();
   }
}
```

1. ***При этом мы должны указывать наш метод может вызвать исключительную ситуацию***

Итоги

1. *Коллекции*
2. *List, ArrayList*
3. *Set, HashSet*
4. *Map, HashMap*
5. *Queue, LinkedList*
6. *Метод add()*
7. *Метод isEmpty()*
8. *Generics*
9. *Error*
10. *Exception*
11. *throw*
12. *try*
13. *catch*

Спасибо!