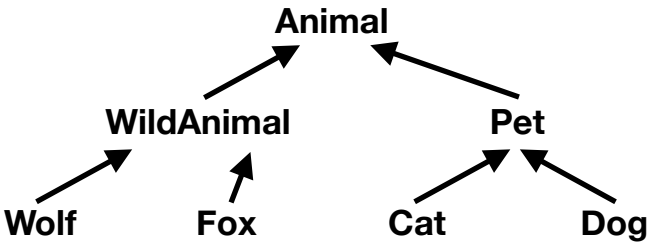
1. Приведение типов при наследовании

Приведение типов - это преобразование значения переменной одного типа в значение другого типа.

Рассмотрим следующую иерархию:



Класс Cat является производным от класса Pet, который, в свою очередь является производным от класса Animal. Таким образом, такая запись будет верна:

```
Animal animalCat = new Cat();
Animal animalDog = new Dog();
```

Пример выше - это расширяющее (или неявное) приведение: ссылки animalCat и animalDog указывают на объекты Cat и Dog. Через эти ссылки можно получить доступ к любым методам, которые есть в классе Animal, но нельзя получить доступ к специфическим методам классов Cat и Dog.

Сужающее (явное) приведение происходит в другую сторону:

```
Cat cat = (Cat) animalCat;
Dog dog = (Dog) animalDog;
```

Сужающее приведение не всегда возможно, при этом компилятор не укажет на ошибку, но в RunTime будет сгенерирован ClassCastException. Такая ситуация возникнет, например, при преобразовании Cat в Dog:

```
Dog dog = (Dog) animalCat;
```

Для того, чтобы избежать exception, необходимо использовать instanceof:

```
if (animalCat instance Dog) {
        Dog dog = (Dog) animalCat;
}
```

Если объект, на который указывает ссылка animalCat, не является объектом класса Dog, то преобразование не произойдет и exception сгенерирован не будет.

Расширяющее преобразование используется вместе с динамическим связыванием - мы можем создать массив ссылок на Animal, которые будут указывать на разных животных в иерархии, а затем для каждой ссылки вызовем метод sleep, который определен в базовом классе. При этом, мы не сможем для такой ссылки вызвать метод eatRabbit, определенный в классе Fox, чтобы не заставить кота есть кроликов.

2. Перечислите методы класса Collections. Укажите назначение этих методов.

Collections - это класс состоящий из статических методов, осуществляющих различные служебные операции над коллекциями. К ним можно обращаться так:

```
Collections.method(Collection);
```

Для работы с любой коллекцией:

frequency(Collection, Object)	возвращает кол-во вхождений элемента в коллекции
disjoint(Collection, Collection)	true, если в коллекциях нет общих элементов
addAll(Collection, T[])	добавляет все элементы массива в коллекцию
min(Collection)	минимальный элемент коллекции
max(Collection)	максимальный элемент коллекции

Для работы со списками:

fill(List, Object)	заменяет каждый элемент в списке значением
sort(List)	сортирует слиянием за O(n*log n)
reverse(List)	изменяет порядок всех элементов

rotate(List, int)	передвигает все элементы на заданный диапазон
binarySearch(List, Object)	ищет элемент (бинарный поиск)
shuffle(List)	перемешивает элементы в случайном порядке
copy(List dest, List src)	копирует один список в другой
replaceAll(List, Object old, Object new)	заменяет все вхождения одного значения на другое
swap(List, int, int)	меняет местами элементы на указанных позициях
indexOfSubList(List src, List trg)	индекс первого вхождения списка trg в список src
lastIndexOfSubList(List src, List trg)	индекс последнего вхождения списка trg в список src

Создание копий коллекции:

newSetFromMap(Map)	создает Set из Map
unmodifiableCollection(Collection)	создает неизменяемую копию коллекции
synchronizedCollection(Collection)	создает потокобезопасную копию коллекции
asLifoQueue(Deque)	создает очередь last in first out из Deque
<T> Set<T> singleton(T o)	создает неизменяемый Set с заданным объектом (только с ним)

---

3. Повторный выброс исключения.

При разработке кода возникают ситуации, когда в приложении необходимо инициировать генерацию исключения, например, для того, чтобы указать на некорректность параметров. Для генерации исключительной ситуации и ручного создания экземпляра исключения используется оператор throw. Экземпляром исключения может быть объект Throwable или его подклассов. Имеется два способа получения Throwable-объекта: использование параметра в предложении catch или создание объекта с помощью операции new:

```
1.
try {
   \\код, в котором может возникнуть исключение
} catch (ArrayIndexOutOfBoundsException e) {
   throw e;
}

2.
if (//условие) {
   throw new IllegalArgumentException();
}
```

При достижения оператора throw генерируется исключение, выполнение кода прекращается и ищется ближайший подходящий блок catch:

```
   private void throwEx(int a) throws IOException {
      if (a%2 == 0) {
         throw new IllegalArgumentException();
      } else {
         throw new IOException();
      }
   }

   private void printSmth() {
      try {
         throwEx(3);
      } catch (IllegalArgumentException e) {
         System.err.println("iae");
      } catch (IOException e) {
         System.err.println("ioe");
```

```
    }
  }
```

/* Output:
printSmth.catch

Если метод генерирует исключение с помощью оператора throw и при этом блок catch в методе отсутствует, как в методе throwEx() на примере выше, то для передачи обработки исключения вызывающему методу printSmth() тип проверяемого исключения должен быть указан в операторе throws при объявлении метода: private void throwEx(int a) throws IOException.

Для непроверяемых исключений, являющихся подклассами класса RuntimeException, как IllegalArgumentException на примере выше, throws в объявлении может отсутствовать, так как играет только информационную роль.

Перехваченное исключение может быть выброшено снова. Такая ситуация и называется повторным выбросом исключения. Например, блок catch отлавливает исключения типа IOException и выбрасывает их вновь.

```
  try{
    // some operations
  } catch(IOException e) {
    throw e;
  }
```

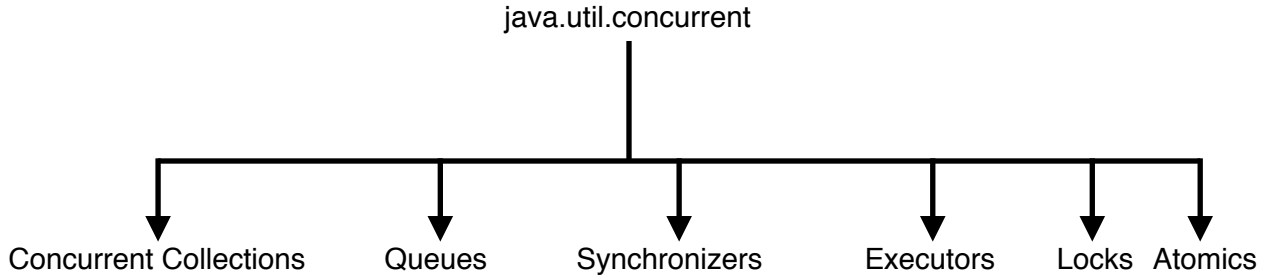
В Java 7 была введена возможность передавать «вверх» по стеку вызова исключение более точного типа, если данные типы указаны при объявлении метода.

```
public static void ioMethod() throws IOException {
  try{
    throw new IOException ("this is IOException ");
  } catch(Exception e) {
    throw e;
  }
}
```

---

4. Классы синхронизированных коллекций пакета java.util.concurrent

В версии Java 5 добавлен пакет java.util.concurrent, классы которого обеспечивают высокую производительность при построении потокобезопасных приложений.



Concurrent Collections - набор коллекций, которые работает намного эффективней в многопоточной среде нежели стандартные коллекции из java.util пакета. Вместо блокирования доступа ко всей коллекции используются блокировки по сегментам данных или же оптимизируется работа для параллельного чтения данных по wait-free алгоритмам. Некоторые из них:

ConcurrentHashMap	аналог Hashtable, данные представлены в виде сегментов, доступ блокируется по сегментам

ConcurrentLinkedQueue	аналог LinkedList
CopyOnWriteArrayList	аналог ArrayList
CopyOnWriteArraySet	имплементация интерфейса Set, за основу взят CopyOnWriteArrayList
ConcurrentSkipListMap	аналог TreeMap
ConcurrentSkipListSet	аналог TreeSet

Queues - неблокирующие и блокирующие очереди с поддержкой многопоточности: ConcurrentLinkedQueue, ConcurrentLinkedDeque, BlockingQueue, ArrayBlockingQueue, BlockingDeque и др.

Synchronizers - вспомогательные утилиты для синхронизации потоков:

Semaphore	предлагает потоку ожидать завершения действий в других потоках
Exchanger	обмен объектами между двумя потоками

Executors - содержит в себе фрейморки для создания пулов потоков, планирования работы асинхронных задач с получением результатов:

Future	интерфейс для получения результатов работы асинхронной операции
ExecutorService	интерфейс, который описывает сервис для запуска Runnable или Callable задач
Executor	организует запуск пула потоков и службы из планирования

Locks - представляет собой альтернативные и более гибкие механизмы синхронизации потоков по сравнению с базовыми synchronized, wait, notify, notifyAll: Lock, ReadWriteLock.

Atomics - классы с поддержкой атомарных операций над примитивами и ссылками.

---

5. Статические методы, особенности работы со статическими методами.

Статические методы являются методами класса, не привязаны ни к какому объекту и не содержат указателя this на конкретный экземпляр, вызвавший метод. Статические методы реализуют парадигму «раннего связывания», жестко определяющую версию метода на этапе компиляции. По причине недоступности указателя this статические методы не могут обращаться к нестатическим полям и методам напрямую, так как они не «знают», к какому объекту относятся, да и сам экземпляр класса может быть не создан.

Для объявления статических методов перед их объявлением используется ключевое слово static. Вызов статического метода всегда следует осуществлять с помощью указания на имя класса, а не объекта. Переопределение статических методов невозможно, так как статические методы связываются во время компиляции, а не в runtime, то есть полиморфизм на статические методы не распространяется.

Статические методы можно импортировать при помощи ключевых слов import static, после чего к статическим методам можно будет обращаться без указания имени класса:

```
import static java.lang.System.out;
public class StaticImport {
	public static void main(String[] args) {
		out.println("hi");
	}
```

```
}
```

Статические методы следует использовать, когда не нужен доступ к состоянию объекта, а все параметры задаются явно, или когда методу нужен доступ только к статическим полям класса.

---

6. Selection Sort, принцип и краткое описание алгоритма

Принцип

На каждом i-ом шаге алгоритма находится минимальный элемент в неотсортированной части массива и меняется местами с i-ым элементом массива.

Алгоритм

на каждом i-ом шаге алгоритма (всего n шагов алгоритма):

1) находим индекс минимального элемента среди всех неотсортированных элементов;
2) меняем местами элемент с найденным индексом и i-ый элемент;

Свойства

- время работы: лучшее – O(n^2), среднее – O(n^2), худшее – O(n^2);
- затраты памяти – O(1);
- неустойчивая;
- количество обменов – обычно O(n) обменов;

Плюсы

- проста в реализации;
- не нужна дополнительная память;
- не больше O(n) обменов;

Минусы

- не ускоряется на частично отсортированных массивах;
- неустойчивая;
- медленная;

---

7. Типы памяти в Java.

В связи с тем, что Java постоянно развивается, на данный момент существует ряд противоречий в терминологии относительно типов памяти в Java. Например, часть ресурсов термины non-heap и stack приравнивают друг к другу. В своем ответе я разделяю память в Java на три области: stack, heap, non-heap.

Stack (Стек):

- имеет небольшой размер в сравнении с кучей, размер стека ограничен, его переполнение может привести к возникновению исключительной ситуации;
- содержит только локальные переменные примитивных типов и ссылки на объекты в куче;
- стековая память может быть использована только одним потоком, т.е. каждый поток обладает своим стеком;
- работает по принципу LIFO(last-in-first-out), благодаря чему работает быстро;
- когда в методе объявляется новая переменная, она добавляется в стек, когда переменная пропадает из области видимости, она автоматически удаляется из стека, а эта область памяти становится доступной для других стековых переменных.

Heap (Куча):

- куча имеет больший размер памяти, чем стек (stack);
- используется для выделения памяти под объекты, объекты хранятся в куче;
- объекты кучи доступны разным потокам;
- в куче работает сборщик мусора: освобождает память, удаляя объекты, на которые нет ссылок.

Non-heap делится на 2 части:

- Permanent Generation – это пул памяти, который содержит интернированные строки, информацию о метаданных, классах, это пространство зарезервировано для классов, статических данных и т.п.
  Metaspace – замена Permanent Generation в Java 8. Основное различие в том, что Metaspace может динамически расширять свой размер во время выполнения. Размер Metaspace по умолчанию не ограничен.
- Code Cache – используется JVM при JIT-компиляции, здесь кэшируется скомпилированный код.