Part I
Fly 2

dedicated to Lera Bublicova

1. Input, Output

1.1. Ввод-вывод №1

InputStream, OutputStream, Reader, Writer.
Стримы - преобразуют что-то в набор байт.
Ридеры/Райтеры - преобразуют что-то в набор чаров.
InputStreamReader- ридер из инпутстрима, то есть хрень, которая преобразует байты в чары
OutputStreanWriter - райтер в аутпутстрим, то есть хрень, которая преобразует чары в байты (это нужно для того, чтобы руководить кодировками)
Фильтрующие потоки - оборачивают какой-то поток внутри себя
Buffered* - обертка нед потоком / ридером-райтером. По сути чтение/запись происходит в буффер, когда буффер обнуляется/перепол тогда считывается/записывается новый блок и помещается в буффер/ буффер обнуляется.

1.2. Ввод-вывод №2

Эмуляция чтения/записи
ByteArrayInputStream | ByteArrayOutputStream
CharArrayReader | CharArrayWriter
StringReader | StringWriter
Конкатенация потоков
SequenceInputStream(InputStream, InputStream)
SequenceInputStream(Enumeration)
PrintWriter | PrintStream - умеет выводить строки, строки с переносом, форматированно выводить строки.
checkError() - проверить была ли ошибка(то есть PrintWriter работает с подавлением ошибок)
DataInputstream | DataOutputStream
Платформонезависимый ввод-вывод примитивных типов и строк. writeT, writeUnsignedByte, writeUnsignedShort, writeUTF - выводит строку в кодировке UTF

Файлы
File - дескриптор.
Методы: getName, getPath, getAbsolutePath, getAbsoluteFile(абсолютный дескриптор) - ситуация когда он нужен: сериализовали файл(File) и передали куда-то. Теперь где-то там нас интересует абсолютный путь, и он у нас есть. String getParent, File get-ParentFile.
Методы: isFile, isHidden, isDirectory, exist, length, lastModified.
Есть проблемы с кроссплатформенностью. Если доступ к файлу закрыт на самом компьютере, либо в настройках java - машины, тогда поведение java-машины на разных ОС может быть разным. В частности, например, типы бросаемых исключений.
Методы: mkdir, mkdirs, createNewFile, delete, deleteOnExit, renameTo(переименовать/перенести в другое место), String[] list, File[] listFiles, String[] list(FileNameFilter), File[] listFiles(FileFilter), canReadFile, canWriteFile, canExecuteFile.

2. Generics

2.1. Мотивация, плюсы, минусы, возможности

Мотивация generic-ов. Чтобы вместо ошибок исполнения получать ошибки компиляции. То есть, например, до 5ой джавы был ArrayList, хранящий Object, и при попытке прикастовать эти Object-ы к стрингу могла возникнуть ошибка. Теперь же, если мы хотим хранить стринги, то пишем ArrayList<String>, и теперь Integer или что-нибудь другое в него просто не добавить.
Плюсы дженериков: один класс на любой тип объекта + не требуется преобразование + защита от неверных типов
C++: отдельный машинный код для каждого типа параметра.

Java: один байт-код на всех. По сути при компиляции тип T заменяется на Object, но везде, где в методах принимается / возвращается тип T, происходит насильный каст к типу параметру, который задан при компиляции. Отсюда и получаются ошибки компиляции: прикастовать Integer к String, например.

Нельзя делать присваивание List<Object> = List<Integer>, например(проблема, видимо, в добавлении).

SuppressWarnings("unchecked") - компилятор не будет ругаться на уничтожение / добавление информации о типе.

Проблемы generic-ов:

T x = new T(); - так нельзя. (все заменяется на Object(ну если нет wildcard-ов)

new T[10] - тоже нельзя, но только для того, чтобы программист не думал, что создает массив типа T.

List<?>[] l = new List<?>[10]; - тоже нельзя. Компилятор тупой, не поймет что там к чему нужно кастовать во время компиляции при вызове, например, l[0].add();

2.2. Wildcard

Wildcard - это ?. Нужны, для того, чтобы в аргумент функции, например, можно было передать нечто, параметризованное совершенно любым типом. Потому что если foo(Collection<Object>), то сюда нельзя передать Collection<String>.

Но в коллекции параметризованные ?, нельзя добавлять ничего, очевидно. По факту, вайлдкарды просто разрешают работу с коллекциями неизвестных типов, но при этом запрещают добавление на уровне компиляции. То есть это продолжение истории о том, что нельзя присваивать List<Integer> li = (List<Object>) lo, которое было в принципе запрещено. Но на самом деле, такое присваивание может быть полезно иногда, и вайлдкарды для этого и созданы.

Bounded wildcard.

Взаимосвязь между типами указывается перед сигнатурой метода!!

Идеология super:

```
<T extends Comparable<T>> max(Collection<T>)
class Kek extends Comparable<Object>
```

Не сработает, если вызвать max(Collection<Kek>)

Поэтому стоит писать:

```
<T extends Comparable<? super T>> max(Collection<T>)
```

3. Внутренние и вложенные классы

3.1. Внутренние(Inner)

Неявно хранят ссылку на объект внешнего класса.

Пример использования. Приватная реализация итератора во внутреннем классе коллекции. В самой коллекции стоит геттер: метод итератор, который возвращает интерфейс. То есть реализация скрыта от пользователя, но пользоваться им как интерфейсом он может.

Анонимные классы.

Синтаксис создания внутреннего класса во внешнем мире.

```
Sequence seq = new Sequence();

Sequence.SequenceSelector sq = seq.new SequenceSelector();
```

Анонимные классы

Когда создаем их в цикле, то создается всего один экземпляр, который имеет ссылки на внешние переменные.

до 8ой джавый все локальные переменные, которые используются в анонимных классах должны быть объявлены final. После 8ой он сам следит за ее финальностью. Если переменная финальная, то можно просто перенести ее на хип и запомнить ссылку.

Локальные классы

Не обязательно анонимные, просто классы, которые объявляются в методах, могут иметь конструктор, имеют доступ ко всему внешнему и локальным final или effective-final переменным.

3.2. Вложенные (Nested)

Ключевое слово static

public - видим его за пределами класса обертки. Обращение Kek.Lol

Есть доступ ко всем статик элементам. Есть доступ ко всем элементам, если есть экземпляр.

Использование:

1. Сокрытие кода.
2. Можно использовать для тестирования приватных методов.

Фабричный метод

Синглтон

Обычная реализация - приватный конструктор и метод getInstance, одно статическое поле instance.

Другой вариант. Сделать nested класс, который будет иметь статическое поле instance, которое должно проинициализироваться в момент статической инициализации.

3.3. Эмулирование множественного наследования

Зачем все это делать??

Важная причина - эмулирование множественного наследования.

Что нам позволяет подобная реализация: кастовать к родителям, вызывать все методы родителей, пользоваться общими переменными в реализации внутренних классов(и не только их). Что как раз и объясняет, почему эти классы должны быть внутренними.

4. Collections

4.1. Методы

contains(Object), remove(Object), add(E), addAll(Collection<? extends E>) - потому что такая коллекция не кастуется к Collection<E>, clear, removeAll(Collection<?> c), retainAll(Collection<?» c), size(), isEmpty(), containsAll(Collection<?>), Object[] toArray(), T[] toArray(T[] a)

UnsupportedOperationException - закрытая операция. То есть конкретно у этой коллекции ее нельзя вызывать(например, она неизменяемая, а хотим вызвать add).

AbstractCollection - унаследовались и получили быструю реализацию.

Неизменяемые коллекции: iterator и size

Изменяемые: еще add и iterator.remove()

4.2. Итераторы

Iterator<E> Collection<E>.iterator();

hasNext() - наличие следующего

next() - следующий

remove() - удаление предыдущего

По факту итератор находится в каждый момент между элементами и перепрыгивает через них.

NoSuchElementException - вызвали next в конце

ConcurrentModificationException - используем этот итератор после того, как изменили коллекцию как-то по-другому (не remove эти итератором)

Интерфейс Iterable

Улучшенный фор, метод iterator().

5. Множества

5.1. Про equals

interface Set<E> extends Collection<E>

Коллекция без повторяющихся элементов.

Требования для Object.equals:

Рефлексивность, транзитивность, симметричность, устойчивость(если объекты не менялись, то результат equals не менялся), o1.equals(null) == false

Наследование и equals => проблемы (пример с Point и ColorPoint)

Симметричный нетранзитивный equals для ColorPoint выглядит так:

```
public boolean equals(Object o) {
        if ( o instanceof ColorPoint) {
                ColorPoint that = (ColorPoint) o;
                return that.x == this.x && that.y == this.y && that.c == this.c;
        }
```

```
        return super.equals(o);
}
```

Варианты решения: 1) сравнивать только одинаковые классы 2) послойное сравнение, то есть вызов super.equals 3) комбинированный вариант с canEquals

5.2. Про HashSet

Множество на основе Object.hashCode
o1.equals(o2) ⇒ o1.hashCode() == o2.hashCode()

6. List

6.1. Методы

subList - кусочек листа, последний невключительно. Ничего не копируется, все действия с sublist отражаются на листе, и наоборот.

6.2. ListIterator

ListIterator<E> extends Iterator<E>
listIterator()
Может ходить как вперед, так и назад:
next(), previous(), nextIndex(), previousIndex(), hasNext(), hasPrevious(), remove(), set(E), add(E)
AbstractList
Неизменяемые списки: get, size
Изменяемые: еще set
Переменной длины: add, remove

7. Queue, Deque

7.1. Queue

Свойства:
1. Порядок выдачи определяется конкретной реализацией
2. Очереди не могут хранить null.
3. У очереди может быть ограничен размер
4. Могут не принять элемент

add, element, remove бросают исключения
offer, peek, poll не бросают, возвращают null, поэтому очереди не согут хранить null.
Реализации очередей
1) LinkedList
2) AbstractQueue: надо реализовать методы size(), offer(E e), peek(), poll(), iterator()

7.2. Deque

те же методы (с исключениями и без) что и в очереди, но + еще на конце слово Last|First
Реализации: ArrayDeque, LinkedList

8. Отображения

8.1. Методы

V get(K), put(K, V), remove(K), containsKey(K), containsValue(V), size(), isEmpty()
Set<Map.Entry<K, V> > entrySet()
Collection<V> values()
Set<K> keySet()
putAll(Map<? extends K, ? extends V> map)
Map.Entry : getKey(), getValue, setValue
Реализации
HashMap - отображение на основе hashCode

LinkedHashMap - сохранение порядка обхода
AbstractMap : достаточно реализовать entrySet().

9. Упорядоченные коллекции

интерфейсы Comparable(метод int compareTo(E e)), Comparator(метод int compare(E e1, E e2))
интерфейс SortedSet(есть методы first, last, headSet, tailSet, subSet)
Его реализует TreeSet.
интерфейс NavigableSet<E>(есть методы higher, ceiling, lower, и т.д)
Его по-прежнему реализует TreeSet
То же самое есть для мапы (SortedMap, etc.)

Part II
Fly 3

dedicated to Lera Abricosova

10. Java 8

10.1. Интерфейсы

Возможность писать статические методы

В 9ой джаве появилась возможность писать private методы

можно писать дефолтную реализацию, но она не доступна для методов equals, hashCode, toString.

Теперь когда есть дефолтные методы, если имплементить много интерфейсов, где возникает коллизия сигнатур функций, заданных дефолтно, то необходимо обязательно указывать реализацию этой функции.

Если коллизия extend vs implement, то выигрывает extend.

Важно понимать, что эти методы с дефолтной реализацией туповатые, они не могут никак подействовать на состояние объекта, так как интерфейс не хранит состояние.

Функциональные интерфейсы - ровно один метод. Function, BiFunction, Consumer, Callable, Supplier

Consumer и Supplier, один из пустого множества в T, другой наоборот.

Есть специальные функциональные интерфейсы типо IntPredicate и т.д, то есть специализация для примитивных типов. Работает быстрее(нет Boxing, Unboxing)

10.2. Lambda

Новый компактный синтаксис для инстанцирования функциональных интерфейсов.

Компилятор сам выводит типы, фигурные скобки не нужны, если внутри одно выражение.

Ссылки на методы

Integer::parseInt - статический метод, параметры лямбды переходят в параметры метода.

Integer::floatValue - нестатический метод, первый параметр - это сам объект, оставшиеся переходят в параметры метода.

Effectively final: ровно одно присваивание, может быть без модификатора final, могут использоваться в лямбда выражениях.

В лямбдах нельзя!

Захват не-final переменных. Ссылки на методы, которые пробрасывают исключения.

10.3. Необязательные значения

Optional<T>. Хранит либо значение типа T, либо ничего.

get() - получить значение, если есть, иначе исключение.

isPresent() - есть ли значение

Статические методы для создания Optional: empty - пустой Optional, of(T t) - ненулевое значение, ofNullable - возможно, нулевое значение.

Зачем он нужен?

Чтобы нагляднее и понятнее обрабатывать случаи null-ов. Например вместо разбора случаем можно написать orElse(”EMPTY”), и тогда, если результат некой операции - это Optional<String>, то выведется то, что мы хотим.

Еще есть orElseThrow

10.4. Сравнения

Comparator получил новые методы:

comparing (еще есть специализация для примитивных)

thenComparing - лексикографический порядок

Эти методы принимают либо (Function), либо (Function, Comparator), который сравнивает возвращаемые значения фукнции

11. Стримы

11.1. Общее

Набор элементов, обрабатываемых оптом.

Классы Stream, IntStream, LongStream, DoubleStream

Может не хранить элементы, может быть ленивым и бесконечным

Получение

Из коллекций и массивов, генераторы, строки из файлов, наборы файлов, случайные потоки.

Все ковейерные операции ленивые. В конце обязательно должен быть терминальный вызов

11.2. Конвейерные методы

filter - фильтрует по предикату, skip - пропускает первые n элементов

distinct - возвращает стрим без дубликатов, peek - применяет какое-то действие

map - преобразует каждый элемент стрима

flatMap - сглаживает все полученные стримы

Например flatMapToInt принимает функцию, которая возвращает стрим интов.

11.3. Терминальные методы

findFirst - первый элемент стрима - возвращает Optional.

findAny - может быть полезен в случае parallelStream чтобы не ждать завершения работы с первой частью.

count - количество элементов.

boolean anyMatch, noneMatch, allMatch, min, max. Min и max возвращают Optional.

forEach - применяет функцию к каждому объекту стрима. Порядок при параллельном выполнении не гарантируется. forEachOrdered

toArray - принимает конструктор массива

12. Collectors

12.1. Редукция

Три операции

1. Определение контейнера. Constructor : Supplier
2. Добавление элемента к контейнеру Accumulator: Function (C, V) -> C
3. Совмещение двух частичных контейнеров Combiner: Function (C, C) -> C

12.2. Collector как интерфейс

Collector <T, A, R> - интерфейс

Supplier<A> supplier

BiConsumer<A, T> accumulator

BinaryOperator<A> combiner

Function<A, R> finisher

12.3. Collectors как класс

Все эти методы возвращают коллектор

joining() - возвращает коллектор, который тупо все соединяет (например строки конкатенирует в одну)

toCollection(Collection::new) - вместо Collection пишем реализацию коллекции, которую хоти получить.

groupingBy(Function f): f(t) - ключ для T t. То есть в итоге получается Map<R, List<T> >, где R - тип возращаемого значения функции.

groupingBy(Function f, Collector downstream) - вместо листа применяем к элементам по этому ключу некоторый коллектор другой.

groupingBy(Function f, Supplier s, Collector downstream) - если мы хотим не HashMap, а что-нибудь другое, TreeMap, допустим.

summingInt - применяет функцию из элемента стрима в инт, суммируем (опять-таки, возвращает коллектор)

mapping - принимает на вход функцию к элементам стрима, применяет, вторым аргументом принимает некоторый другой коллектор для получившегося стрима.

reducing - как reduce

toList, toSet, toMap(keyMapper, valueMapper)

joining(delimiter) - с разделителем

13. Reflection

13.1. Методы

Хотим получить информацию о классах, загруженных в джава-машину.
Class.forName проводит статическую инициализацию.
Обращение к static final полям не проводит
Kek.class не проводит
Обращение к static полям проводит.
Определение типов. Есть такие методы:
isAnnotation, isEnum, isArray, isInterface, is*Class, isPrimitive, isAnonymousClass, isLocalClass isMemberClass
Методы получения мест, где определен класс.
getDeclaredClass, getPackage, getEnclosingConstructor, getEnclosingMethod.
Приведение типов. isAssignableFrom(class), isInstance(object), cast(object), object instanceof class - возвращает boolean.

13.2. Структура Класса

Интерфейс Member
getDeclaringClass - получить класс, в котором определен, getName - получить имя члена, getModifiers - получить класс Modifiers, у которого есть методы isPublic, isFinal итд.

Field
getFields, getDeclaredFields, getField(String name)
getName, getType, get(object), set(object, value), get*(object)
геттеры / сеттеры нужны для сериализации/десериализации, например

Method
getMethods, getDeclaredMethods
getMethod(name, Class... parameters) - конкретный метод
NoSuchMethodException
getName, getParameterTypes, getExceptionTypes, getReturnType
invoke(object, Object... args)
Нужно передавать обертки над примитивными типами
InvocationTargetException

Конструкторы
newInstance(Object... args) - создает новый объект. Возвращаемый тип - параметр типа.
Еще можно class.newInstance() - дефолтный конструктор

Классы и интерфейсы
getClasses - открытые, getDeclaredClasses - все.

Доступ к закрытым членам
isAccessible, setAccessible(boolean)
Массивы
newInstance(Class, length)
newInstance(Class, dim[])
isArray, get

13.3. ClassLoader

Загружает и определяет классы
loadClass(name, resolve) - если передать false, то статической инициализации не будет
findLoadedClass(name) - найти загруженный класс
resolveClass(class) - загружает библиотеки
getClassLoader() - кто загрузил класс, Thread.getContextClassLoader - контекстный загрузчик.
Прямая загрузка класса
forName(name, init, ClassLoader)
URLClassLoader

```
URL jar = new URL("file://.");
className = "Test"
ClassLoader cl = new URLClassLoader(new URL[]{jar});
Class c = cl.loadClass(className);

Method m = c.getMethod("main", String[].class);
m.invoke(null, (Object) new String[]{"hello"});
```

Пример использования. Есть какой-то архиватор, который поддерживает только Хаффмана. При этом иногда полезны другие алгоритмы архивации. Для этого чтобы не лезть в чужой код, можно написать свой Archiver, откомпилировать, положить в папку plugins. Тогда здесь, когда будет вызвана загрузка всех Archiver'ов из директории plugins, мы можем свободно ими пользоваться. В общем, это сделано, чтобы не переписывать/расширять старый код. А каждую новую версию архиватора писать и компилировать отдельно.

Reflection API ОЧЕНЬ МЕДЛЕННО РАБОТАЕТ!!1