Экзамен Java

Билет 1

Объект (object) – это самоописывающая структура данных, обладающая внутренним состоянием и способная обрабатывать передаваемые ей сообщения.

Инкапсуляция (incapsulation) – один из основных принципов объектно-ориентированного программирования, заключающийся в том, что доступ к внутреннему состоянию объекта извне осуществляется только через механизм передачи сообщений.

Класс (class) – это тип данных, значениями которого являются объекты, имеющие сходное внутреннее состояние и обрабатывающие одинаковый набор сообщений.

К **членам класса** относятся:

- экземплярные поля - обеспечивают хранение внутреннего состояния объектов;
- статические поля - предназначены для хранения данных, общих для всех объектов класса;
- экземплярные методы - отвечают за обработку передаваемых объекту сообщений;
- статические методы - выполняют действия, для которых не нужен доступ к конкретному объекту класса;
- экземплярные конструкторы - инициализируют только что созданные объекты класса;
- статический конструктор (набор static-блоков) - инициализирует статические поля класса;
- вложенные классы - главным образом, представляют объекты, необходимые для реализации данного класса.

```
public class MyClass {

}
```

Билет 2

Экземплярное поле (instance field) – именованная составная часть внутреннего состояния объекта.

Статическое поле (static field), принадлежащее некоторому классу – это поле, разделяемое всеми объектами этого класса.

Билет 3

Экземплярный метод (instance method) – это подпрограмма, осуществляющая обработку переданного объекту сообщения.

Статический метод (static method), объявленный в некотором классе – это метод, не имеющий доступа к внутреннему состоянию объектов этого класса.

```
class Point {
    public double x, y;

    public double dist() {
        return Math.sqrt(x*x + y*y);
    }

    public static boolean less(Point a, Point b) {
        return a.dist() < b.dist();
```

```
    }
}
```

Перегрузка метода (method overloading) – это объявление для заданного класса двух или более методов, имеющих одинаковое имя, но различные сигнатуры.

Позднее связывание (late binding) – это определение адреса вызываемого экземплярного метода на основе информации о классе объекта во время выполнения программы.

Экземплярные методы, для вызовов которых выполняется позднее связывание, называются **виртуальными** (virtual methods).

Билет 4

Экземплярный конструктор (instance constructor) – это экземплярный метод, предназначенный для инициализации только что созданного объекта.

Конструктор по умолчанию (default constructor) – это экземплярный конструктор с пустой сигнатурой.

```
class Point {
    private double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

Билет 5

В Java объекты, в отличие от значений примитивных типов, могут располагаться только в динамической памяти (куче). Кроме того, объекты не могут вкладываться друг в друга. Другими словами, объекты не могут лежать в локальных переменных и параметрах методов, в полях объектов и элементах массивов. Вместо этого там хранятся только указатели на объекты, которые в языке Java называются объектными ссылками.

Создание объекта в Java выполняется с помощью операции `new`:

```
new имя_класса(фактические_параметры_конструктора)
```

Операция new имеет специальную форму для создания инициализированного массива (массивовый литерал):

```
new тип_элемента [] { значение , значение , ... }
```

Билет 6

Статический конструктор (static constructor) – это статический метод, предназначенный для инициализации статических полей класса. Статический конструктор не имеет параметров и вызывается до вызова любого статического метода или конструктора класса.

В Java объявление класса может содержать один или несколько «static»-блоков, записывемых как

```
static {
    последовательность_операторов
}
```

«Static»-блоки выполняются при первом обращении к классу, где под обращением к классу мы будем понимать создание объекта, вызов статического метода, обращение к статическому полю и т.д.
Если объявление класса содержит несколько «static»- блоков, то они будут выполняться в том порядке, в котором они перечислены в теле класса. Совокупность всех «static-блоков» класса играет роль статического конструктора.

Билет 7

Тип данных A является **подтипом** (subtype) для типа данных B, если программный код, расчитанный на обработку значений типа B, может быть корректно использован для обработки значений типа A.

Субтипизация (subtyping) – это свойство языка программирования, означающее возможность использования подтипов в программах.

Различают неявную и явную субтипизацию.
В языке программирования, поддерживающем неявную субтипизацию, решение о том, является ли тип A подтипом для типа B, принимается на основе анализа структуры значений этих типов. Поэтому **неявную субтипизацию** часто называют структурной субтипизацией (structural subtyping).
В языке программирования, поддерживающим **явную субтипизацию** (explicit subtyping или nominal subtyping), тип A является подтипом для типа B тогда и только тогда, когда A является наследником B.
В языке Java используется явная субтипизация.

Билет 8

Если новый класс A создан на основе уже существующего класса B с помощью механизма наследования, то говорят, что класс A является **производным классом** (derived class) или подклассом (subclass) по отношению к классу B. В свою очередь, класс B выступает в роли **базового класса** (base class) или суперкласса (superclass) для класса A.

Тело любого конструктора производного класса должно начинаться с **вызова одного из конструкторов базового класса.** В Java вызов конструктора базового класса осуществляется из тела конструктора производного класса с помощью оператора `super`:

Операция динамического **приведения объектной ссылки** obj к типу T проверяет, является ли тип T одним из типов объекта obj, и возвращает obj, если является. В противном случае операция порождает исключение ClassCastException. В Java динамическое приведение записывается как `(тип)obj`.

Переопределение метода (method overriding) – это замена тела экземплярного метода базового класса в производном классе, позволяющая объекту производного класса по-другому обрабатывать то же самое сообщение.

Билет 9

Абстрактный метод (abstract method) – это не имеющий тела виртуальный метод, который должен быть переопределён в производных классах.

Абстрактный класс (abstract class) – это класс, имеющий абстрактные методы, которые либо объявлены в нём самом, либо унаследованы от базовых классов и не переопределены.

Билет 10

Интерфейс (interface) – это тип данных, представляющий собой набор абстрактных методов.

```
interface имя {
    ...
}
```

Реализация интерфейса (interface implementation) – это класс, являющийся подтипом этого интерфейса. Если этот класс – неабстрактный, то в нём должны быть определены все без исключения методы интерфейса.

Интерфейс может быть наследником других интерфейсов, которые перечисляются в его объявлении после ключевого слова extends.

Билет 11

Класс X называется **экземплярным вложенным классом** (inner class) в классе Y, если из методов класса X доступно внутреннее состояние объекта класса Y. При этом класс Y называется **объемлющим классом**.

Класс X называется **статическим вложенным классом** (static nested class) в классе Y, если из методов класса X доступны статические поля класса Y.

Локальный класс (local class) – экземплярный вложенный класс, объявленный внутри метода объемлющего класса и имеющий доступ к неизменяемым локальным переменным и параметрам этого метода.

Анонимный класс (anonymous class) – локальный класс, объявление которого совмещено с созданием его экземпляра.

Билет 12

Функциональный интерфейс (functional interface) – интерфейс с единственным абстрактным методом.

Создание экземпляра анонимного класса, реализующего функциональный интерфейс, синтаксически может быть представлено в виде лямбда-выражения, имеющего две формы:

1. (форм. параметры) -> тело_метода
2. (форм. параметры) -> выражение

Билет 13

Контейнерный класс – это класс, объекты которого выступают в роли хранилищ других объектов.

Формальный типовый параметр – это тип, который используется в объявлении обобщённого класса и при этом неизвестен во время компиляции обобщённого класса.

Обобщённый класс – это класс, имеющий формальные типовые параметры.

Билет 14

Формальный типовый параметр – это тип, который используется в объявлении обобщённого класса и при этом неизвестен во время компиляции обобщённого класса.

Обобщённый класс – это класс, имеющий формальные типовые параметры.

Ограниченный обобщённый класс – это обобщённый класс, хотя бы для одного формального типового параметра которого задана верхняя граница.

Билет 15

В Java массивы **ковариантны**. Это означает, что если класс S – наследник класса T, то тип S[] является подтипом T[]. В частности, ковариантность массивов позволяет написать такой код:

```
Integer[] ints = new Integer [10];
Number[] numbers = ints;
```

Дело в том, что обобщённые классы в Java **инвариантны**, т.е. если G – обобщённый класс, и класс S – наследник класса T, то классы `G<S>` и `G<T>` отношением наследования не связаны.

Шаблон обобщённого класса G – это неявный супертип для любого класса, порождённого из G. Шаблон получается параметризацией класса G специальным фактическим параметром «?». Например, `Stack<?>`.

Осоьенности:

- Можно объявить переменную типа шаблон, но невозможно создать его объект.
- Если тип переменной – шаблон некоторого обобщенного класса G, то этой переменной можно присвоить ссылку на любой объект класса G, независимо от того, какой фактический типовый параметр был передан классу G при создании объекта.

Билет 16

Если обобщённый класс имеет несколько формальных типовых параметров, то возможно получить **частичный шаблон**, передав классу «?» вместо части фактических типовых параметров.

Верхнюю границу «?» в шаблоне можно уточнить, используя следующую запись:

```
G<..., ? extends X, ...>
```

У шаблона, ограниченного сверху, недоступны методы, тип параметров которых соответствует «?» в шаблоне

Билет 17

По умолчанию «?» в шаблонах ограничен сверху, однако можно, наоборот, оганичить его снизу, записав

```
G<..., ? super X, ...>
```

У шаблона, ограниченного снизу, недоступны методы, тип возвращаемого значения которых соответствует «?» в шаблоне.

Билет 18

Обобщённый метод – это статический или экземплярный метод класса, имеющий формальные типовые параметры.

В объявлении метода типовые параметры непосредственно предшествуют типу возвращаемого значения метода. Вызов статического обобщённого метода записывается как

```
имя_класса.<факт. типовые параметры>имя_метода(...)
```

Вызов экземплярного обобщённого метода:

```
объект.<факт. типовые параметры>имя_метода(...)
```

Билет 19

Нештатная ситуация – это ситуация, в которой выполнение некоторого фрагмента кода программы оказывается по тем или иным причинам невозможно.

Перехват нештатных ситуации – это механизм, обеспечивающий продолжение работы программы при возникновении нештатной ситуации.

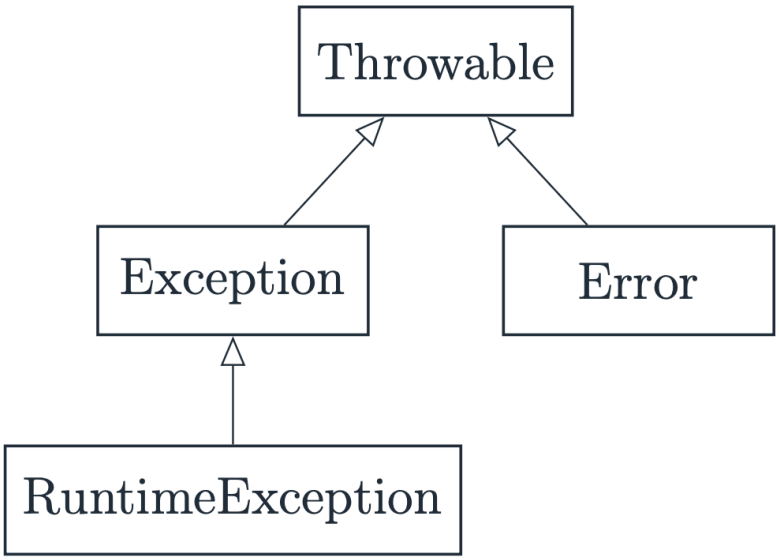
Два механизма перехвата нештатных ситуаций:

- **Обработка кодов возврата**. Функция, во время выполнения которой может возникнуть нештатная ситуация, возвращает некоторое значение, которое говорит о том, успешно или неуспешно функция выполнила свою задачу. Перехват ситуации заключается в том, что в коде, вызывающем такую функцию, стоят проверки ее возвращаемого значения.
- **Обработка исключений.** В случае возникновения нештатной ситуации генерируется так называемое исключение, описывающее нештатную ситуацию. Генерация исключения приводит к передаче управления на фрагмент кода программы, называемый обработчиком исключения.

Билет 20

Исключительная ситуация – это нештатная ситуация, возникшая в силу внешних по отношению к программе причин.

Определение 40. *Исключение* – это объект, описывающий нештатную ситуацию.



`Throwable` – базовый класс для всех классов исключений.
`Error` – базовый класс для классов исключений, описывающих «смертельные» для программы системные ошибочные ситуации (переполнение стека и т.п.).
`Exception` – базовый класс для всех классов несистемных исключений.
`RuntimeException` – базовый класс для классов несистемных исключений, описывающих ошибочные ситуации.

Билет 21

Порождённое и неперехваченное исключение приводит к аварийному завершению программы с выдачей дампа стека вызовов, записанного в исключении.

```
try {
    /* код, в котором может возникнуть нештатная ситуация */
    ...
} catch (SomeException e) {
    /* обработчик исключений, которые можно привести к типу SomeException */
    ...
} catch (SomeException2 e) {
```

```
    ...
}
... /* другие catch-блоки */...
finally {
    /* код, который должен вызываться при любом выходе из try-блока */
    ...
}
```

Билет 22

Порождение исключения выполняется оператором `throw` исключение; Например,

```
throw new SomeException();
```

Если класс исключения не является подклассом классов Error и RuntimeException, то для любого порождённого исключения в коде программы должен быть предусмотрен обработчик. Это требование синтаксически выражается следующим образом: если в процессе выполнения некоторого метода m могут порождаться исключения x, y и z, и эти исключения в теле метода не перехватываются, то объявление метода должно выглядеть как

```
тип m(параметры) throws x, y, z
{
    ...
}
```

Билет 23 и 24

Функтор – это обобщённый класс, параметризованный типом T и инкапсулирующий значение(-я) типа T, удовлетворяющий следующим условиям:

- объекты функтора – неизменяемые;
- у функтора имеется метод map, получающий в качестве параметра функцию (т.е. лямбда-выражение) преобразования значений типа T;
- метод map применяет функцию к значению(-ям), лежащему(-им) внутри функтора, и возвращает новый объект функтора, в котором инкапсулирован(-ы) результат(-ы) применения функции;
- если передать методу map функцию, которая просто возвращает значение своего аргумента (функция-идентичность), то единственным эффектом вызова метода map будет порождение объекта функтора, идентичного исходному объекту

Реализация метода map и forEach в функторе Roots

```
public class Roots<T> {
    ...
    public <R> Roots<R> map(Function<T, R> f) {
        HashSet<R> c = new HashSet <>();
        for (T t : container) c.add(f.apply(t));
        return new Roots<>(c);
    }
    public void forEach(Consumer<T> f) {
        for (T t : container) f.accept(t);
    }
    public <R> Roots<R> flatMap(Function<T, Roots<R>> f) {
        HashSet<R> c = new HashSet <>();
        map(f).forEach(rs -> c.addAll(rs.container));
        return new Roots<>(c);
```

```
    }
    ...
}
```

```
Roots<Roots<Double>> roots = Roots.of(a, b, c, 1e-10).map(y -> Roots.of(1.0, p, -y, 1e-10));
Roots<Double> roots = Roots.of(a, b, c, 1e-10).flatMap(y -> Roots.of(1.0, p, -y, 1e-10));
```

Билет 25

Монада – это функтор с дополнительным методом flatMap, удовлетворяющий следующим условиям:

- метод flatMap получает в качестве параметра функцию, способную преобразовать каждое значение внутри монады и завернуть результат преобразования в новый объект-монаду;
- метод flatMap применяет функцию ко всем значениям, хранящимся внутри монады, и объединяет получившиеся объекты-монады в один объект;
- метод flatMap ассоциативен, т.е. выражение `m.flatMap(f).flatMap(g)` эквивалентно выражению `m.flatMap(x -> f(x).flatMap(g))`

Объект монады `Optional<T>` является контейнером для нуля или одного объекта класса T. Подразумевается, что объект `Optional<T>` находится в одном из двух состояний:

- является пустым;
- содержит ненулевую ссылку на объект класса T.

Создание объекта `Optional<T>`:

- создание пустого объекта: `static <T> Optional<T> empty()`
- заворачивание ненулевой объектной ссылки: `static <T> Optional<T> of(T value)`
- заворачивание объектной ссылки с порождением пустого объекта, если ссылка – нулевая: `static <T> Optional<T> ofNullable(T value)`

«Развернуть» объект `Optional<T>` можно с путём вызова метода get:

Пусть необходимо выполнить композицию частичных функций

$$f(g(h(x)))$$

Здесь каждая функция – это вызов некоторого метода. Так как функции частичные, то они не определены для некоторых значений своих аргументов, т.е. могут возвращать нулевую ссылку. Но при этом каждая функция подразумевает, что передаваемое ей значение существует, т.е. выражается ненулевой ссылкой.

Билет 26

```
public static Optional<String> compose(HashMap<String, Integer> a,
                                       HashMap<Integer, Float> b,
                                       HashMap<Float, String> c,
                                       String x) {
    return Optional.ofNullable(x)
                   .map(a::get)
                   .map(b::get)
                   .map(c::get);
}
```

Билет 27

Интерфейс `Stream<T>` в каком-то смысле является развитием идеи итераторов, т.е. представляет последовательность некоторых значений, называемую потоком. Получить объект `Stream<T>` можно из объекта любого контейтерного класса стандартной библиотеки языка Java путём вызова метода `stream`:

```
Stream<T> stream()
```

Вытащить все значения из потока `Stream<T>` можно с помощью метода forEach:

```
void forEach(Consumer<? super T> action)
```

Этот метод получает в качестве параметра лямбда-выражение и вызывает его для каждого значения из последовательности, представляемой потоком.

Методы интерфейса **Stream<T>** позволяют преобразовывать последовательности:

- формирование потока, содержащего только те элементы исходного потока, которые удовлетворяют предикату:

```
Stream<T> filter(Predicate<? super T> predicate)
```

- формирование потока, представляющего последовательность значений, получаемую путём вызова функции для каждого элемента исходного потока:

```
<R> Stream<R> map(
    Function<? super T, ? extends R> mapper)
```

- конкатенация потоков, получаемых путём вызова функции для каждого элемента исходного потока:

```
<R> Stream<R> flatMap(
    Function<? super T,
             ? extends Stream<? extends R>> mapper
)
```

Чтобы продемонстрировать применение преобразования `flatMap`, возьмём целочисленную матрицу `ArrayList<ArrayList<Integer>>` и распечатаем последовательность её ненулевых элементов:

```
public static void nonZeroes(
        ArrayList<ArrayList<Integer>> m) {
    m.stream()
            .flatMap(row ->
                row.stream().filter(x -> x != 0)
            )
            .forEach(System.out::println);
}
```

Потребление значений из потоков выполняется так называемыми *коллекторами* – объектами классов, реализующими интерфейс `Collector`. Коллекторы предназначены для выполнения анализа потоков и записи результатов реализуемых потоками преобразований в объекты контейнерных классов.
Чтобы задействовать некоторый коллектор, следует вызвать метод `collect` потока:

```
<R,A> R collect(Collector<? super T,A,R> collector)
```

Билет 28

Пакет – это контейнер классов, предоставляющий для них отдельное пространство имён и дополнительные возможности по управлению доступом.

Для создания пакета нужно поместить в начало каждого файла, содержащего исходный текст классов, которые вы планируете включить в проект, директиву

```
package имя_пакета;
```

Классы, объявленные в файлах, в которых отсутствует директива package, считаются принадлежащими безымянному пакету по умолчанию (default package).
Компилятор Java требует, чтобы файлы пакета размещались в файловой системе в каталоге, путь к которому относительно текущего каталога (в котором запущен компилятор) соответствовал имени пакета.

Например, файлы пакета ru.bmstu.iu9 должны находиться в каталоге ru/bmstu/iu9. Файлы, принадлежащие пакету по умолчанию, должны находиться в текущем каталоге.

Напомним, что один java-файл может содержать не более одного публичного класса и произвольное количество непубличных классов.
Публичные классы можно использовать как внутри, так и снаружи пакета, а непубличные классы доступны из любого места внутри пакета, но извне не видны.
К членам классов пакета, объявленным без модификаторов доступа, можно обращаться из любого места внутри пакета, но запрещено обращаться снаружи пакета.

Билет 29

Для того чтобы обратиться к классу, принадлежащему другому пакету, этот класс нужно импортировать. Существует три способа импорта класса.

1. Использование квалифицированного имени. Например: `java.util.HashSet<String> a = new java.util.HashSet<>();`
2. Указание класса в директиве import:

```
import java.util.HashSet; // в начале файла`
...
HashSet<String> a = new HashSet <>();
```

4. Использование директивы import для импорта всех классов пакета:

```
import java.util.*; // в начале файла
...
HashSet<String> a = new HashSet <>();
```