ООР

1. Определение инкапсуляции, наследования, полиморфизма

Инкапсуляция – принцип, позволяющий объединить данные и код, манипулирующий этими данными, а также защищающий данные от прямого внешнего доступа и неправильного использования, т.е. данные сокрыты, а доступ к ним возможен лишь посредством методов данного класса. В Java закрытие полей с помощью модификатора доступа private обеспечивает инкапсуляцию. А для работы с сокрытыми данными принято организовывать специальные методы: геттеры (get) и сеттеры (set) (обычно с public-доступом согласно концепции JavaBeans).

Наследование – принцип, позволяющий одному классу наследовать поля и методы другого класса и добавлять к ним свои поля и методы, характерные только для него. Дочерний класс обычно добавляет свои поля и методы к уже имеющимся полям и методам класса-родителя. Поэтому производный класс обычно более специфичный и представляет меньшую группу объектов.

Наследование бывает:

- одиночное – когда производный класс может иметь только одного предка;
- множественное – класс может иметь любое количество предков (в Java для классов множественное наследование запрещено, но есть возможность имплементировать несколько интерфейсов, а также сами интерфейсы могут наследовать свои характеристики от нескольких интерфейсов).

Полиморфизм – принцип, согласно которому реализуется механизм использования одного и того же имени метода для решения похожих, но несколько отличающихся задач в различных объектах при наследовании от одного суперкласса. Полиморфизм позволяет использовать одно и тоже имя для решения двух или более схожих, но различно реализованных задач.

Обеспечивающими полиморфизм механизмами являются механизм «позднего связывания» и механизм «переопределения методов».

Позднее связывание происходит в процессе выполнения программы, определяет принадлежность объекта к конкретному классу и производит вызов метода именно этого класса. Этот механизм позволяет определить версию полиморфного метода во время выполнения программы. То есть на этапе компиляции не всегда возможно определить, какая версия переопределенного метода будет вызвана на этапе выполнения программы. При вызове метода сначала он ищется в самом классе, при его наличии он вызывается, если его нет, то метод с данной сигнатурой ищется и вызывается в родительском классе, если его нет и там, то поиск продолжается вверх по иерархическому дереву наследования вплоть до корня иерархии (родителя всех объектов в Java – класса Object).

Метод подкласса переопределяет метод суперкласса, если его сигнатура полностью совпадает с сигнатурой метода суперкласса. **Переопределение методов** реализует полиморфизм с помощью позднего связывания. Когда переопределенный метод вызывается через ссылку суперкласса, во время выполнения определяется, какую версию метода вызвать, основываясь на типе объекта, на который имеется ссылка. То есть, во время выполнения тип объекта определяет версию вызываемого метода. Таким образом, реализуется возможность работать с несколькими классами так, будто это один и тот же класс, но в тоже время поведение каждого класса будет уникальными в зависимости от реализации.

2. Определение раннего и позднего связывания, механизма переопределения методов

Механизм связывания устанавливает, какой именно метод будет вызван при использовании ссылки на объект. Связывание может быть ранним и поздним.

Раннее связывание, основываясь на **типе ссылки**, устанавливает на этапе компиляции, какой метод будет вызван, т.к. необходимая информация для определения, какой именно метод будет вызван, известна на этапе компиляции. В Java реализуют раннее связывание: перегрузка методов, статические методы.

Позднее связывание происходит в процессе выполнения программы, определяет принадлежность объекта к конкретному классу и производит вызов метода именно этого класса. Этот механизм позволяет определить версию полиморфного метода во время выполнения программы. То есть на этапе компиляции не всегда возможно определить, какая версия переопределенного метода будет вызвана на этапе выполнения программы. При вызове метода

сначала он ищется в самом классе, при его наличии он вызывается, если его нет, то метод с данной сигнатурой ищется и вызывается в родительском классе, если его нет и там, то поиск продолжается вверх по иерархическому дереву наследования вплоть до корня иерархии (родителя всех объектов в Java – класса Object).

Метод подкласса переопределяет метод суперкласса, если его сигнатура полностью совпадает с сигнатурой метода суперкласса. **Переопределение методов** реализует полиморфизм с помощью позднего связывания. Когда переопределенный метод вызывается через ссылку суперкласса, во время выполнения определяется, какую версию метода вызвать, основываясь на типе объекта, на который имеется ссылка. То есть, во время выполнения тип объекта определяет версию вызываемого метода. Таким образом, реализуется возможность работать с несколькими классами так, будто это один и тот же класс, но в тоже время поведение каждого класса будет уникальными в зависимости от реализации.

3. Приведение типов при наследовании

Приведение типов – это преобразование значения переменной одного типа в значение другого типа. **Приведение типов при наследовании** – это приведение ссылочных типов.

Приведение ссылочных типов можно классифицировать следующим образом:
восходящее (расширяющее) и нисходящее (сужающее);

При **восходящем** (восходящее по иерархии наследования) приведении типов подкласс приводится к суперклассу, то есть мы приводим более специфицированный класс к более общему, который описывает большее количество объектов, поэтому это приведение типов называется **расширяющим**. **Восходящее** приведение типов является безопасным, т.к. это переход от более конкретного типа к более общему, то есть функционал подкласса ограничивается функционалом суперкласса. Поскольку подклассу доступен весь функционал суперкласса, данное приведение типа является **безопасным**. В связи с безопасностью данного приведения типов в Java его можно проводить **неявно**:

```
String string = new String();

Object object = (Object) string; // явное приведение типов – допустимо, но избыточно

Object object = string; // неявное приведение типов – допустимо
```

Использование неявного восходящего приведения типов, когда ссылка типа суперкласса указывает на дочерний класс, позволяет пользоваться преимуществами полиморфного вызова. Например, в цикле foreach перебираются ссылки Object на различные объекты и вызывается метод toString(), который переопределен в классах, к которым принадлежат эти объекты, вследствие чего метод toString() создает различные строки, опираясь на реализацию этого метода в каждом из классов.

При **нисходящем** (нисходящее по иерархии наследования) приведении типов суперкласс приводится к подклассу, то есть мы приводим более общий тип к более специфицированному, который описывает меньшее количество объектов, поэтому это приведение типов называется **сужающим**. **Нисходящее** приведение типов **не является безопасным**, т.к. это переход от более общего типа к более конкретному, а более конкретный тип может содержать функционал, который отсутствует в суперклассе, вследствие чего могут возникнуть ошибки. Так как данное приведение опасно его нужно проводить **явным** образом, то есть нужно прописывать приведение типов в коде:

```
Object object = new Object();

String string = object; // неявное приведение типов – ошибка компиляции

String string = (String) object; // явное приведение типов – ошибка времени выполнения
```

В последнем примере видно, что мы приводим object к типу String, который не является типом String, при этом компилятор на эту ошибку никак не отреагирует, а вот во время выполнения мы получим исключение типа **ClassCastException**. Чтобы избежать этого можно использовать ключевое слово **instanceof** для проверки типов:

```
Object object = new Object();

if (object instanceof String) {

    String string = (String) object;
```

}

4. Наследование: преимущества и недостатки, альтернатива.

Наследование – принцип, позволяющий одному классу наследовать поля и методы другого класса и добавлять к ним свои поля и методы, характерные только для него. Дочерний класс обычно добавляет свои поля и методы к уже имеющимся полям и методам класса-родителя. Поэтому производный класс обычно более специфичный и представляет меньшую группу объектов.

При объявлении в подклассе полей, совпадающих по сигнатуре с полями суперкласса, они не переопределяются и существуют в одном объекте независимо друг от друга. При этом для обращения к полям экземпляра текущего класса можно использовать ключевое слово **this**, а для обращения к полю текущего экземпляра родительского класса можно использовать ключевое слово **super**. Ключевое слово **super** выступает в роли ссылки на текущий экземпляр родительского класса, его можно использовать для обращения к конструкторам родительского класса, методам и полям текущего экземпляра родительского класса. Ключевое слово **this** выступает в роли ссылки на текущий экземпляр класса, его можно использовать для обращения к методам, полям текущего экземпляра класса, а также для доступа к перегруженным конструкторам данного класса. При вызове конструкторов с помощью **this** и **super** нужно учитывать, что эти конструкции вызываются в именно в конструкторах, должны быть единственными и первыми конструкциями в вызывающем их конструкторе, не могут вызываться вместе в одном конструкторе.

Наследование в Java никогда не убирает в потомке поля или методы суперкласса.

Наследование в Java реализуется с помощью ключевого слова **extends**:

class Cat extends Animal {}

В Java можно запретить наследование того или иного класса, объявив его **final**:

final class Cat {}

Наследование бывает:

- одиночное – когда производный класс может иметь только одного предка;
- множественное – класс может иметь любое количество предков (в Java для классов множественное наследование запрещено, но есть возможность имплементировать несколько интерфейсов, а также сами интерфейсы могут наследовать свои характеристики от нескольких интерфейсов).

Главное преимущество наследования – расширение существующего функционала без дублирования кода.

Главный недостаток наследования – сильная связанность подкласса с суперклассом. В связи с чем могут возникнуть ситуации, которые повлекут непредвиденные последствия в подклассе при изменении суперкласса. Как было сказано выше, наследование в Java никогда не убирает в потомке поля или методы суперкласса. Часть из этих полей и методов может быть не нужна подклассу или представлять собой опасность, но убрать их из него возможности нет.

В целом наследование следует разумно применять при отношениях между классами **IS A** (Cat **IS A**n Animal) для построения понятной иерархии классов. Но в архитектурах с большими и громоздкими иерархиями классов вышеуказанные недостатки наследования начинают проявляться особенно активно.

Использование наследования можно заменить композицией и агрегацией, то есть связью **HAS A**. В ряде случаев это даже выглядит более естественно с точки зрения логики. Например, есть класс Engine, представляющий собой двигатель. Можно создать Car extends Engine, но все же машина не является именно двигателем, скорее двигатель является частью машины. И в данном случае лучше подойдет отношение **HAS A**: Car **HAS A**n Engine.

И агрегация, и композиция реализуют связь **HAS A**, которая выражается в том, что один класс содержит поля, которые являются его составными частями, при этом класс может делегировать своим частям то или иное поведение, ту или иную обработку переменных. Разница между композицией и агрегацией заключается в том, что композиция является более сильной связью и подразумевает, что класс-целое контролирует время жизни своей составной части, то есть вне объекта класса-целого данная составная часть не существует. Агрегация же подразумевает более слабую связь, чем композиция, не связывая время жизни составной части и целого (например, составная часть передается в целое через параметры конструктора).

Наилучшим образом уменьшить связность объектов можно с помощью использования именно **агрегации**. При ее использовании можно добиться достаточной гибкости, чтобы она служила хорошей альтернативой наследованию.

5. Инкапсуляция

Инкапсуляция – принцип, позволяющий объединить данные и код, манипулирующий этими данными, а также защищающий данные от прямого внешнего доступа и неправильного использования, т.е. данные сокрыты, а доступ к ним возможен лишь посредством методов данного класса. В Java закрытие полей с помощью модификатора доступа private обеспечивает инкапсуляцию. А для работы с сокрытыми данными принято организовывать специальные методы: геттеры (get) и сеттеры (set) (обычно с public-доступом согласно концепции JavaBeans).

6. Полиморфизм, механизмы Java, его обепечивающие

Полиморфизм – принцип, согласно которому реализуется механизм использования одного и того же имени метода для решения похожих, но несколько отличающихся задач в различных объектах при наследовании от одного суперкласса. Полиморфизм позволяет использовать одно и тоже имя для решения двух или более схожих, но различно реализованных задач.

Обеспечивающими полиморфизм механизмами являются механизм «позднего связывания» и механизм «переопределения методов».

Позднее связывание происходит в процессе выполнения программы, определяет принадлежность объекта к конкретному классу и производит вызов метода именно этого класса. Этот механизм позволяет определить версию полиморфного метода во время выполнения программы. То есть на этапе компиляции не всегда возможно определить, какая версия переопределенного метода будет вызвана на этапе выполнения программы. При вызове метода сначала он ищется в самом классе, при его наличии он вызывается, если его нет, то метод с данной сигнатурой ищется и вызывается в родительском классе, если его нет и там, то поиск продолжается вверх по иерархическому дереву наследования вплоть до корня иерархии (родителя всех объектов в Java – класса Object).

Метод подкласса переопределяет метод суперкласса, если его сигнатура полностью совпадает с сигнатурой метода суперкласса. **Переопределение методов** реализует полиморфизм с помощью позднего связывания. Когда переопределенный метод вызывается через ссылку суперкласса, во время выполнения определяется, какую версию метода вызвать, основываясь на типе объекта, на который имеется ссылка. То есть, во время выполнения тип объекта определяет версию вызываемого метода. Таким образом, реализуется возможность работать с несколькими классами так, будто это один и тот же класс, но в тоже время поведение каждого класса будет уникальными в зависимости от реализации.

7. Модификаторы доступа

В Java есть 4 модификатора доступа, обеспечивающих возможность настройки области видимости: **private, по умолчанию (package-private), protected, public**.

private – члены класса или классы (вложенные) доступны только в самом классе;

по умолчанию – члены класса или классы доступны в том пакете, в котором находятся (любые члены класса и сами классы без указания модификаторов доступа имеют этот доступ, для обозначения этого доступа не нужно использовать ключевых слов);

protected – члены класса или классы доступны в том пакете, в котором находятся, а также в наследниках класса, находящихся в других пактах;

public – члены класса или классы доступны везде;

Действие модификатора доступа распространяется только на тот класс/член класса, перед которым он указан.

Любые члены класса, внутренние и вложенные классы могут иметь любой из 4 вышеуказанных модификаторов доступа. Внешние классы (классы, реализация которых не находится в других классах) могут иметь только один из двух модификаторов доступа: **public** или **по умолчанию**.

Во время наследования возможно изменение модификаторов доступа в сторону большей видимости. Например, можно переопределить **protected-метод**, объявив его **public**.

8. Что такое объектно-ориентированный подход

ООП – подход программирования, основанный на представлении программы в виде совокупности объектов, каждый из которых является экземпляром конкретного класса.

Класс – это тип данных, который обобщенно описывает набор родственных объектов, состоит из набора полей (переменных других типов данных) и методов (описывают поведение родственных объектов, имеют доступ к полям класса).

Объект является конкретным экземпляром класса. Любой объект относится к определенному классу. Объектом называется модель реальной сущности, обладающая конкретными значениями свойств и поведением. Объект представляет собой именованный набор данных, которые находятся в памяти компьютера, и методов.

Например, существует класс Flat, который описывает поля (address, floor, area, owner) и методы (build(), repair() и т.п.). А объектом этого класса уже будет являться определенная квартира с именем flatOnNevsky и конкретными значениями полей (address = “St.-Petersburg, Nevsky prospect, 150, 13”, floor = 3, area = 54, owner = “Svetlana”).

ООП основано на следующих принципах:

Абстракция данных – определение значимых характеристик сущности, определяющих ее концептуальные границы и отличающих ее от других сущностей.

Инкапсуляция – принцип, позволяющий объединить данные и код, манипулирующий этими данными, а также защищающий данные от прямого внешнего доступа и неправильного использования, т.е. данные сокрыты, а доступ к ним возможен лишь посредством методов данного класса. В Java закрытие полей с помощью модификатора доступа private обеспечивает инкапсуляцию. А для работы с сокрытыми данными принято организовывать специальные методы: геттеры (get) и сеттеры (set) (обычно с public-доступом согласно концепции JavaBeans).

Наследование – принцип, позволяющий одному классу наследовать поля и методы другого класса и добавлять к ним свои поля и методы, характерные только для него.

Наследование бывает:

- одиночное – когда производный класс (подкласс) может иметь только одного предка (суперкласс);
- множественное – класс может иметь любое количество предков (в Java для классов множественное наследование запрещено, но есть возможность имплементировать несколько интерфейсов, а также сами интерфейсы могут наследовать свои характеристики от нескольких интерфейсов).

Полиморфизм – принцип, согласно которому реализуется механизм использования одного и того же имени метода для решения похожих, но несколько отличающихся задач в различных объектах при наследовании от одного суперкласса. Полиморфизм позволяет использовать одно и тоже имя для решения двух или более схожих, но различно реализованных задач.

Обеспечивающими полиморфизм механизмами являются механизм «позднего связывания» и механизм «переопределения методов».

Позднее связывание происходит в процессе выполнения программы, определяет принадлежность объекта к конкретному классу и производит вызов метода именно этого класса. Этот механизм позволяет определить версию полиморфного метода во время выполнения программы. То есть на этапе компиляции не всегда возможно определить, какая версия переопределенного метода будет вызвана на этапе выполнения программы. При вызове метода сначала он ищется в самом классе, при его наличии он вызывается, если его нет, то метод с данной сигнатурой ищется и вызывается в родительском классе, если его нет и там, то поиск продолжается вверх по иерархическому дереву наследования вплоть до корня иерархии (родителя всех объектов в Java – класса Object).

Метод подкласса переопределяет метод суперкласса, если его сигнатура полностью совпадает с сигнатурой метода суперкласса. **Переопределение методов** реализует полиморфизм с помощью позднего связывания. Когда переопределенный метод вызывается через ссылку суперкласса, во время выполнения определяется, какую версию метода вызвать, основываясь на типе объекта, на который имеется ссылка. То есть, во время выполнения тип объекта определяет версию вызываемого метода. Таким образом, реализуется возможность работать с несколькими классами так, будто это один и тот же класс, но в тоже время поведение каждого класса будет уникальными в зависимости от реализации.

Java API

1. Создание, модификация и сравнение объектов класса String. Пул литералов.

Объектом класса **String** является строка. Объект типа **String** может быть создан с помощью оператора **new** с вызовом определенного конструктора или с помощью литерала, заключенного в двойные кавычки. Строки являются неизменяемыми (**immutable**), то есть строковое значение не может быть изменено после создания объекта-строки с помощью какого-либо метода. Любое изменение приводит к созданию нового объекта-строки. Таким образом, ссылка типа **String** ссылается на строку-константу. Ссылка типа **String** может быть изменена таким образом, что будет ссылаться на другой объект типа **String**, но сам объект является неизменным. Класс **String** в Java объявлен **final**, то есть создание собственных классов-наследников со всеми характеристиками String невозможно.

```
String s = “string”; // создание строки
String s = new String(“string”); // создание строки
String s = “”; // создание пустой строки
String s = new String(); // создание пустой строки
String s = null; // пустая ссылка, которая не указывает ни на какую строку
```

Некоторые конструкторы класса String:

```
String()
String(String str)
String(char[] value)
String(char[] value, int offset, int count)
String(StringBuilder builder)
String(StringBuffer buffer)
```

Сравнение строк не следует проводить с помощью оператора == (!=), т.к. в таком случае будут сравниваться ссылки объектов, а не сама последовательность символов, что может привести, например, к false-результату сравнения идентичных по значению строк.

Для сравнения строк у класса String есть определенные методы:

boolean equals(Object obj) и **boolean equalsIgnoreCase(String s)** – сравнение строк с учетом регистра и без учета регистра соответственно;

int compareTo(String s) и **int compareToIgnoreCase(String s)** – лексикографическое сравнение строк с учетом и без учета регистра соответственно. Возвращает 0, если строки идентичны друг другу, значение меньше нуля – если строка меньше строки-аргумента, значение больше нуля – если строка больше строки-аргумента.

Пул литералов – это коллекция ссылок на строковые объекты. Строки являются частью пула литералов и размещены в куче, а сами ссылки на них находятся в пуле литералов.

При создании строки с помощью двойных кавычек, сначала в пуле литералов ищется строка с таким значением, если находится, то возвращается ссылка на нее, если не находится, то создается новая строка в пуле литералов, а после возвращается ссылка на нее.

При создании строки с помощью оператора **new**, мы принуждаем класс **String** создать новый объект строки, независимо от того, есть ли он в пуле литералов или нет. Затем мы можем использовать метод **intern()** для того, чтобы поместить эту строку в пул литералов или получить ссылку из пула литералов на другой объект **String** с таким же значением.

Неизменяемость строк и использование в Java пула литералов тесно связаны между собой. Если бы строки были изменяемы, использование пула литералов утратило бы свой смысл. Можно представить, что будет, если в одной части программы будет ссылка на определенную строку, а в другой части программы кто-то имеет ссылку на эту же строку и изменяет ее по мере необходимости. В таком случае использование пула литералов привело бы к серьезным ошибкам. Несмотря на то, что создание строки может занимать больше времени, сам по себе пул литералов позволяет экономить большой объем памяти, поэтому и было организовано его использование в Java, в связи с чем строки были созданы объектами, которые невозможно изменить.

Пример работы пула литералов (сравнение ссылок):

```
String s1 = "Cat";
String s2 = "Cat";
String s3 = new String("Cat");
```

```
System.out.println("s1 == s2: " + (s1 == s2)); //s1 == s2: true
System.out.println("s1 == s3: " + (s1 == s3)); //s1 == s3: false
s3 = s3.intern();
System.out.println("s1 == s3: " + (s1 == s3)); //s1 == s3: true
```

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

Collections – класс, состоящий из статических методов, осуществляющих различные служебные операции над коллекциями.

Некоторые его методы (22):

- **sort(List)** – сортировать список, используя merge sort алгоритм, с гарантированной скоростью O (n*log n);
- **binarySearch(List, Object)** – бинарный поиск элементов в списке, список должен быть отсортирован;
- **reverse(List)** – изменить порядок элементов в списке на противоположный;
- **shuffle(List)** – случайно перемешать элементы;
- **fill(List, Object)** – заменить каждый элемент заданным;
- **copy(List dest, List src)** – скопировать список src в dst;
- **min(Collection)** – вернуть минимальный элемент коллекции;
- **max(Collection)** – вернуть максимальный элемент коллекции;
- **rotate(List list, int distance)** – циклически повернуть список на указанное число элементов;
- **replaceAll(List list, Object oldVal, Object newVal)** – заменить все объекты на указанные;
- **indexOfSubList(List source, List target)** – вернуть индекс первого подсписка в source, который эквивалентен target;
- **swap(List, int, int)** – поменять местами элементы в указанных позициях списка;
- **unmodifiableCollection(Collection)** – создает неизменяемую копию коллекции. Существуют отдельные методы для Set, List, Map, и т.д.;
- **synchronizedCollection(Collection)** – создает потокобезопасную копию коллекции. Существуют отдельные методы для Set, List, Map, и т.д.;
- **checkedCollection(Collection<E> c, Class<E> type)** – создает типо-безопасную копию коллекции, предотвращая появление неразрешенных типов в коллекции. Существуют отдельные методы для Set, List, Map, и т.д.;
- **<T> Set<T> singleton(T o)** – создает неизменяемый Set, содержащий только заданный объект. Существуют методы для List и Map;
- **<T> List<T> nCopies(int n, T o)** – создает неизменяемый List, содержащий n копий заданного объекта;
- **frequency(Collection, Object)** – подсчитывает количество заданных элементов в коллекции;
- **disjoint(Collection, Collection)** – определяет, что коллекции не содержат общих элементов;
- **addAll(Collection<? super T>, T[])** – добавляет все элементы из массива в коллекцию;
- **newSetFromMap(Map)** – создает Set из Map;
- **asLifoQueue(Deque)** – создает LIFO Queue представление из Deque;

3. Класс Object, методы класса Object

Вершиной иерархии классов в Java является класс **Object**, который является суперклассом для всех других классов. Ссылочная переменная типа **Object** может ссылаться на объект любого другого класса. В классе **Object** определены методы, которые наследуются всеми классами **(10)**:

- **protected Object clone()** – создает и возвращает копию вызывающего объекта. Нужно учитывать, если данный метод не переопределен в других классах, он будет производить неглубокое копирование (поля класса будут ссылаться на те же объекты и т.п.). Кроме этого, если необходимо вызывать его другим классом, то другой класс должен реализовывать tagged-интерфейс **Cloneable**, в противном случае при вызове

данного метода не избежать **CloneNotSupportedException**. Для правильного клонирования следует не только реализовывать интерфейс **Cloneable**, но и явно переопределять метод **clone()**, чтобы он производил глубокое копирование, т.е. все ссылочные поля класса должны ссылаться не на те же самые объекты, что и в исходном объекте, а на новые объекты-клоны, у классов которых тоже должен быть реализован интерфейс **Cloneable** и переопределен метод clone() соответствующим образом;

- **public boolean equals(Object ob)** – предназначен для использования и переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов одного и того же типа;
- **public Class<? extends Object> getClass()** – возвращает объект типа Class;
- **protected void finalize()** – автоматически вызывается сборщиком мусора (garbage collection) перед уничтожением объекта;
- **public int hashCode()** – вычисляет и возвращает хэш-код объекта (число, в общем случае вычисляемое на основе значений полей объекта);
- **public String toString()** – возвращает представление объекта в виде строки.
- final-методы **notify()**, **notifyAll()** и **wait()**, **wait(long timeout)** – предназначены для работы с многопоточностью, позволяют организовать связь между потоками в виде объекта-монитора, у которого вызываются данные методы, когда один поток уведомляет об освобождении объекта-монитора другой/другие объекты **(notify()**/ **notifyAll())**, а другие потоки находятся в режиме ожидания освобождения объекта-монитора **(wait())**.

При переопределении метода **equals()** следует переопределять и метод **hashCode()**, т.к. между собой эти методы должны соблюдать определенную взаимосвязь:

- если объекты равны друг другу по **equals()**, то у них **должен** быть одинаковый хэш-код;
- различные объекты по **equals() могут** иметь разный хэш-код, но это не является обязательным условием, их хэш-код может быть идентичным;

4. Методы класса String

Объектом класса String является строка. Строки являются неизменяемыми (immutable), то есть строковое значение не может быть изменено после создания объекта-строки с помощью какого-либо метода. Любое изменение приводит к созданию нового объекта-строки. Ссылка типа String может быть изменена таким образом, что будет ссылаться на другой объект типа String, но сам объект является неизменным.

Некоторые методы класса String (25)

Методы общего назначения

- **char charAt(int position)** – возвращение символа из указанной позиции (нумерация с нуля);
- **void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** – извлечение символов строки в массив символов;
- **int length()** – определение длины строки;
- **boolean isEmpty()** – возвращает true, если длина строки равна 0;
- **String intern()** – заносит строку в «пул» литералов и возвращает ее объектную ссылку;
- **String concat(String s)** или **«+»** – слияние строк;
- **String toUpperCase()/toLowerCase()** – преобразование всех символов вызывающей строки в верхний/нижний регистр;

Методы сравнения строк

- **boolean equals(Object obj)** и **boolean equalsIgnoreCase(String s)** – сравнение строк с учетом регистра и без учета регистра соответственно;
- **int compareTo(String s)** и **int compareToIgnoreCase(String s)** – лексикографическое сравнение строк с учетом и без учета регистра соответственно. Возвращает 0, если строки идентичны друг другу, значение меньше нуля – если строка меньше строки-аргумента, значение больше нуля – если строка больше строки-аргумента.

- **boolean contentEquals(StringBuffer ob)** – сравнение строки и содержимого объекта типа StringBuffer;

Поиск символов и подстрок

- **int indexOf(char ch)** – определение позиции символа в строке;
- **int indexOf(String str)** – определение позиции первого символа указанной подстроки в строке;
- **boolean endsWith(String suffix)** – заканчивается ли String суффиксом suffix;
- **boolean startsWith(String prefix)** – начинается ли String с префикса prefix;

Извлечение подстрок

- **String trim()** – удаление всех пробелов в начале и конце строки;
- **String substring(int startIndex, int endIndex)** – извлечение из строки подстроки длины endIndex-startIndex, начиная с позиции startIndex. Нумерация символов в строке начинается с нуля;
- **String substring(int startIndex)** – извлечение из строки подстроки, начиная с позиции startIndex до конца строки;

Приведение значений элементарных типов и объектов к строке

- **static String valueOf(*значение*)** – преобразование переменной базового типа к строке;

Форматирование строк

- **static String format(String format, Object… args)**, **static String format(Locale l, String format, Object… args)** – генерирует форматированную строку, полученную с использованием формата, интернационализации и др.;

Сопоставление с образцом

- **String replace(char oldChar, char newChar)** – возвращает строку, где все символы oldChar заменены на newChar;
- **String replace(CharSequence target, CharSequence replacement)** – возвращает строку, заменяя элементы target на replacement.
- **boolean matches(String regexStr)** – проверяет удовлетворяет ли строка указанному регулярному выражению;
- **String replaceFirst(String regexStr, String replacement)** – заменяет первое вхождение строки, удовлетворяющей регулярному выражению, указанной строкой;
- **String replaceAll(String regexStr, String replacement)** – заменяет все вхождения строк, удовлетворяющих регулярному выражению, указанной строкой;
- **String[] split(String regexStr)** – разбивает строку на части, границами разбиения являются вхождения строк, удовлетворяющих регулярному выражению;

5. Работа с объектами типа StringBuilder и StringBuffer

Классы **StringBuilder** и **StringBuffer** по своему предназначению близки к классу **String**, но в отличие от класса String содержимое и размеры объектов классов **StringBuilder** и **StringBuffer** можно изменять.

Основным и единственным различием между **StringBuilder** от **StringBuffer** является потокобезопасность **StringBuffer**. В связи с тем, что **StringBuilder** не является потокобезопасным, работает он быстрее, чем **StringBuffer**. В связи с вышесказанным можно однозначно сделать выбор в пользу использования **StringBuilder** при отсутствии вероятности использования объекта в конкурирующих потоках, а при наличии такой вероятности следует выбирать **StringBuffer**.

Объекты **StringBuilder**, **StringBuffer**, **String** можно преобразовывать друг в друга с помощью соответствующих методов и конструкторов. Конструкторы классов **StringBuilder** и **StringBuffer** могут принимать в качестве параметра объект String.

При создании объекта одного из этих классов конструктор по умолчанию автоматически резервирует объем памяти под 16 символов. Количество символов, под которое необходимо зарезервировать память, можно указать в конструкторе, что бывает удобным, если известно хотя бы приблизительно, до каких масштабов будет разрастаться объект. Если длина строки после изменения превышает зарезервированный размер, то емкость объекта автоматически увеличивается, оставляя некоторый резерв для дальнейших действий.

Если методы объекта класса **StringBuilder** или **StringBuffer** производят изменения содержимого самого объекта, то изменяется текущий объект, а не создается новый объект, как в случае с объектами класса **String**.

Для классов **StringBuilder** и **StringBuffer** не переопределены методы **equals()** и **hashCode()**, то есть сравнивать содержимое двух объектов невозможно, а хэш-коды объектов этих классов вычисляются так же, как и для объектов класса **Object**. При необходимости сравнения содержимого объекты классов **StringBuilder** или **StringBuffer** можно преобразовывать к объекту класса **String** с помощью метода **toString()**, после чего выполнить сравнение с помощью метода **equals()** класса **String**.

Некоторые методы классов **StringBuilder** и **StringBuffer (8)**:

- **void setLength(int n)** – установка размера вместимости;
- **void ensureCapacity(int minimum)** – установка гарантированного минимального размера вместимости;
- **int capacity()** – возвращение текущего размера вместимости;
- **StringBuilder/StringBuffer append(*параметры*)** – добавление к содержимому объекта строкового представления аргумента, который может быть символом, значением базового типа, массивом и строкой;
- **StringBuilder/StringBuffer insert(*параметры*)** – вставка символа, объекта или строки в указанную позицию;
- **StringBuilder/StringBuffer deleteCharAt(int index)** – удаление символа;
- **StringBuilder/StringBuffer delete(int start, int end)** – удаление подстроки;
- **StringBuilder/StringBuffer reverse()** – обращение содержимого объекта.

В классе присутствуют также методы, аналогичные методам класса String, такие как replace(), substring(), charAt(), length(), getChars(), indexOf() и др.

6. Очереди (Queue, Dequeue). Конкретные классы очередей.

Queue – коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и просмотра элементов. Очереди обычно (но не обязательно) упорядочивают элементы по принципу FIFO (first-in-first-out).

Методы интерфейса **Queue<E> (6)**:

- **boolean add(E o)** — вставляет элемент в очередь, если же очередь полностью заполнена, то генерирует исключение IllegalStateException;
- **boolean offer(E o)** — вставляет элемент в очередь, если возможно;
- **E element()** — возвращает, но не удаляет головной элемент очереди;
- **E peek()** — возвращает, но не удаляет головной элемент очереди, возвращает null, если очередь пуста;
- **E remove()** — возвращает и удаляет головной элемент очереди;
- **E poll()** — возвращает и удаляет головной элемент очереди, возвращает null, если очередь пуста.

Методы **element()** и **remove()** отличаются от методов **peek()** и **poll()** тем, что генерируют исключение **NoSuchElementException**, если очередь пуста.

Обычно очередь не может содержать элементы **null**, т.к. **null** используется методами **poll()** и **peek()** в случаях, когда очередь пуста. Тем не менее, в некоторых реализациях интерфейса **Queue** можно добавлять **null** элементы (пример: **LinkedList**).

Интерфейс **Deque** наследует интерфейс **Queue** и определяет двунаправленную очередь, позволяет добавлять, удалять и просматривать элементы с двух концов (начало очереди и конец очереди). Реализация этого интерфейса может быть использована для моделирования стека. Методы, обеспечивающие удаление, вставку и просмотр элементов существуют в двух формах (подобно **Queue**): одни создают исключительную ситуацию в случае неудачного завершения, другие возвращают какое-то значение (**null** или **false**). Методы существуют для работы с очередью с двух сторон: **addFirst(E e)/addLast(E e), getFirst()/getLast(), peekFirst()/peekLast()** и т.д. по аналогии.

Конкретные классы очередей:

- **LinkedList** – класс связного списка, который может использоваться в качестве очереди, т.к. реализует интерфейс **Deque;**
- **ArrayDeque** – класс, представляющий очередь переменного размера с эффективной реализация **Deque**. **ArrayDeque** работает быстрее, чем Stack, если используется в виде стека, и быстрее, чем **LinkedList**, если используется, как очередь;
- **PriorityQueue** – класс очереди с приоритетами, реализует интерфейс **Queue**. По умолчанию очередь с приоритетами размещает объекты согласно возрастающему порядку сортировки с использованием **Comparable**. Элементу с наименьшим значением присваивается высший приоритет. Если элементы одинаковы с точки зрения сравнения, то порядок их расположения относительно друг друга произволен. Если объекты, которые добавляются в очередь, нельзя сопоставить с использованием **Comparable**, то при добавлении 2ого элемента возникает исключительная ситуация **ClassCastException**. В очереди с приоритетами. можно указать специальный порядок размещения, используя **Comparator**.
- Имеется ряд классов-очередей, которые организованы специально для работы в многопоточной среде, например, **ArrayBlockingQueue;**
-

7. Пеопределение метода toString();

Метод toString() следует переопределять таким образом, чтобы кроме или вместо стандартной информации о пакете, в котором находится класс, имени самого класса и его хэш-кода, он возвращал информацию о значимых значениях полей класса, таким образом возвращая полезную информацию о себе в строковом виде. Метод toString() в классе Object возвращает стандартную информацию в виде:

getClass().getName() + '@' + Integer.toHexString(hashCode())

Пример переопределения метода **toString()**:

```
class Cat{
   private String name;
   private int age;

   public Cat(String name, int age) {
      this.name = name;
      this.age = age;
   }

   @Override
   public String toString() {
      return String.format("%s: name = '%s' age = %d",
            getClass().getSimpleName(), name, age);
   }
}
```

8. Основные характеристики списков (List). Конкретные классы списков.

Список (List) – это упорядоченная коллекция, в которой объекты хранятся в порядке их добавления в список, элементы в списке могут повторяться. Интерфейс List сохраняет последовательность добавления элементов и позволяет осуществлять доступ по индексу. Интерфейс List является наследником интерфейса Collection и предоставляет методы в дополнение к методам интерфейса Collection.

Методы интерфейса **List<E> extends Collection<E> (11)**:

- **E get(int index)** – возвращает объект, находящийся в позиции index;
- **E set(int index, E element)** – заменяет элемент, находящийся в позиции index объектом element;
- **boolean add(E element)** – добавляет элемент в список

- **void add(int index, E element)** – вставляет элемент element в позицию index, при этом список раздвигается
- **E remove(int index)** – удаляет элемент, находящийся на позиции index;
- **boolean addAll(int index, Collection<? extends E> c)** – добавляет все элементы коллекции с в список, начиная с позиции index в списке;
- **int indexOf(Object o)** – возвращает индекс первого появления элемента o в списке;
- **int lastIndexOf(Object o)** – возвращает индекс последнего появления элемента o в списке;
- **ListIterator<E> listIterator()** – возвращает итератор на список;
- **ListIterator<E> listIterator(int index)** – возвращает итератор на список, установленный на элемент с индексом index;
- **List<E> subList(int from, int to)** – возвращает новый список, представляющий собой часть данного (начиная с позиции from до позиции to-1 включительно).

Основные классы, реализующие интерфейс **List** это **Stack**, **Vector**, **ArrayList** и **LinkedList**.

ArrayList – класс, представляющий список на базе массива, который может динамически меняться. **ArrayList** позволяет получить любой элемент по его индексу за фиксированное время. Но добавление/удаление элементов в него требует затрат времени пропорциональных размеру, потому что нужно подвинуть все элементы с места вставки/удаления и до конца списка: либо освобождая место для вставляемого элемента, либо убирая пропуск на месте удаленного элемента.

LinkedList – класс, представляющий двусвязный список. хранит элементы в двусвязном списке. **LinkedList** поддерживает добавление/удаление элементов за фиксированное время, но только последовательный доступ к элементам. То есть, можно перебрать список с начала в конец и с конца в начало, но получение элемента в середине списка займет время пропорциональное размеру списка.

LinkedList требует больше памяти для хранения такого же количества элементов, чем **ArrayList**, потому что кроме самого элемента хранятся еще указатели на следующий и предыдущий элементы списка, тогда как в **ArrayList** элементы просто идут по порядку.

Vector – класс, представляющий устаревшую версия **ArrayList**, обладает схожей функциональностью с **ArrayList**, является потокобезопасным, в связи с чем работает медленнее, чем **ArrayList**.

Stack – класс-наследник класса **Vector**, реализующий модель стека с упорядочиванием элементов по принципу LIFO (last-in-first-out). Как и **Vector** является потокобезопасным и устаревшим классом. Вместо класса Stack рекомендуется использовать классы, реализующие интерфейс **Deque**.

Exceptions

1. Множественные catch

В некоторых случаях в одном блоке кода (**try**) может быть сгенерировано несколько типов исключений, поэтому необходимо обеспечить наличие нескольких блоков **catch**, если только блок **catch** не обрабатывает все исключения, которые могут возникнуть блоке **try**.

```
try {
    // some operations
} catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Выход за границы массива");
} catch (IOException e) {
    System.err.println("Ошибка ввода-вывода");
}
```

Когда возникает исключение, каждый оператор **catch** проверяется по порядку и первый из них, чей тип соответствует типу исключения, выполняется. Остальные блоки **catch** игнорируются, а выполнение программы продолжается со строки, следующей за последним блоком **catch**. Поэтому **подклассы** исключений в блоках **catch** должны следовать перед любым из их **суперклассов**, иначе блок **catch**, отлавливающий исключения **суперкласса**, будет перехватывать эти исключения.

```
try {
    // some operations
} catch (Exception e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

В ситуации приведенной выше будет выявлена ошибка компиляции, т.к. второй блок **catch** никогда не сможет быть обработан, т.к. первый блок **catch** с суперклассом **Exception** будет перехватывать исключения подкласса **IOException**.

Зачастую бывают ситуации, когда инструкций catch несколько, а обработка исключений в них идентичная, например, вывод сообщения об исключении. Поэтому в 7 версии Java появилась возможность объединять идентичные обработки блоков catch с помощью использования одного обработчика для нескольких исключений (multi-catch).

Для одновременной обработки нескольких типов исключений типы исключений в блоке **catch** разделяются с помощью оператора **ИЛИ** (**OR**) – «**|**».

```
try {
    // some operations
} catch(NumberFormatException  | ClassNotFoundException e) {
    e.printStackTrace();
}
```

2. Назначение оператора throws

Если метод может породить исключение, которое сам не обрабатывает, он должен передать это исключение вызывающему его методу, чтобы в нем оно было обработано или передано дальше. Передача исключений вызывающему методу в Java обеспечивается добавлением ключевого слова **throws** в заголовок объявления метода.

После ключевого слова **throws** должны быть перечислены через запятую типы исключений, которые метод может выбросить. Это необходимо для всех исключений кроме исключений **RunTimeException**, ошибок **Error** и их подклассов. Все другие исключения, которые могут быть сгенерированы в методе, но не обрабатываются в нем, **должны быть** перечислены после ключевого слова **throws**.

```
String read() throws IOException {
    try (BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));) {
        return reader.readLine();
```

```
    }
  }
```

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;
    }
}
```

3. Блок try-с ресурсами

В Java 7 реализована возможность автоматического закрытия ресурсов в блоке try с ресурсами (**try-with-resources**).

В операторе try открывается ресурс (например, файловый поток ввода), который затем читается. При завершении блока try данный ресурс автоматически закрывается, поэтому нет никакой необходимости явно вызывать метод **close()** у потока, как это было в предыдущих версиях Java.

Автоматическое управление ресурсами возможно только для тех ресурсов, которые реализуют интерфейс java.lang.**AutoCloseable**.

```
try (BufferedReader reader = new BufferedReader(new InputStreamReader(System.in))) {
    reader.readLine();
} catch (IOException e) {
    e.printStackTrace();
}
```

Закрытие ресурсов **до Java 7**:

```
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
try {
    reader.readLine();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (reader != null) {
            reader.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

4. Checked и runtime исключения

Исключительные ситуации типа **Exception** делятся на проверяемые (**checked**) исключения и на **RuntimeException** – непроверяемые (**unchecked**) исключения.

Проверяемые исключения должны быть обработаны в методе, который их генерирует, либо включены в **throws**-список исключений метода для передачи этих исключений в вызывающие методы, которые проведут их дальнейшую обработку. Возможность возникновения проверяемого исключения может быть отслежена на этапе компиляции кода.

Непроверяемые исключения, представленные классом **RuntimeException** и его наследниками, автоматически генерируются при возникновении ошибок во время выполнения программы без каких-либо проверок на этапе компиляции. Компилятор не проверяет, генерируют ли методы эти исключения, как не проверяет и наличие обработки этих исключений.

Некоторые подклассы проверяемых исключений **(8)**:

- **ClassNotFoundException** – класс не найден;
- **CloneNotSupportedException** – попытка клонировать объект, который не реализует интерфейс **Cloneable**;
- **IllegalAccessException** – доступ к классу отклонен;
- **InstantiationException** – попытка создавать объект абстрактного класса или интерфейса;
- **InterruptedException** – один поток был прерван другим потоком;
- **NoSuchFieldException** – требуемое поле не существует;
- **NoSuchMethodException** – требуемый метод не существует;
- **IOException** – общий класс исключений, вызванных прерванными или неудачно завершенными операциями ввода-вывода.

Некоторые подклассы непроверяемых исключений **(15)**:

- **ArithmeticException** – арифметическая ошибка типа деления на нуль;
- **ArrayIndexOutOfBoundsException** – индекс массива находится вне границ;
- **ArrayStoreException** – назначение элементу массива несовместимого типа;
- **ClassCastException** – недопустимое приведение типов;
- **IllegalArgumentException** – при вызове метода использован недопустимый аргумент;
- **IllegalMonitorStateException** – недопустимая операция монитора, типа wait() на мониторе в потоке, который не владеет монитором;
- **IllegalStateException** – среда или приложение находятся в некорректном состоянии;
- **IllegalThreadStateException** – требуемая операция не совместима с текущим состоянием потока;
- **IndexOutOfBoundsException** – некоторый тип индекса находится вне границ;
- **NegativeArraySizeException** – массив создавался с отрицательным размером;
- **NullPointerException** – недопустимое использование нулевой ссылки;
- **NumberFormatException** – недопустимое преобразование строки в числовой формат;
- **SecurityException** – попытка нарушить защиту;
- **StringIndexOutOfBoundsException** – попытка индексировать вне границ строки;
- **UnsupportedOperationException** – встретилась неподдерживаемая операция;

5. Повторный выброс исключения

Программа может сама явно выбрасывать исключения, используя оператор **throw**. Это называется «ручным» выбросом исключения.

Общая форма использования оператора **throw**:

throw **ThrowableInstance**;

ThrowableInstance должен быть объектом типа **Throwable** или подкласса **Throwable**. Примитивные типы или классы, не являющиеся **Throwable** не могут использоваться в качестве исключений.

Получить объект **Throwable** можно двумя способами: использовать объект, полученный в блоке **catch** или создать **Throwable** объект с помощью оператора **new**.

При достижении оператора **throw** выполнение кода прекращается, блок **try/catch** проверяется на наличие соответствующего обработчика **catch**. Если он существует, то управление передается в этот блок **catch**. Создание объекта-исключения с помощью **new** без оператора **throw** не вызывает исключительную ситуацию.

Если метод может породить исключение, которое сам не обрабатывает, он должен передать это исключение вызывающему его методу, чтобы в нем оно было обработано или передано дальше. Передача исключений вызывающему методу в Java обеспечивается добавлением ключевого слова **throws** в заголовок объявления метода. После ключевого слова **throws** должны быть перечислены через запятую типы исключений, которые метод может выбросить, кроме исключений **RuntimeException** и ошибок **Error**.

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;
    }
}
```

6. Особенности работы блока finally

Может возникнуть ситуация, при которой нужно выполнить некоторые действия по завершению программы (например, закрыть поток) вне зависимости от того, было сгенерировано исключение или нет. В этом случае используется блок finally, который обязательно выполняется после инструкций try или catch. Например:

```
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
try {
    reader.readLine();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (reader != null) {
            reader.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

В Java 7 реализована возможность автоматического закрытия ресурсов в блоке try с ресурсами (try-with-resources). В операторе try открывается ресурс (например, файловый поток ввода), который затем читается. При завершении блока try данный ресурс автоматически закрывается, поэтому нет никакой необходимости явно вызывать метод close() у потока, как это было в предыдущих версиях Java. Автоматическое управление ресурсами возможно только для тех ресурсов, которые реализуют интерфейс java.lang.AutoCloseable.

```
try (BufferedReader reader = new BufferedReader(new InputStreamReader(System.in))) {
    reader.readLine();
} catch (IOException e) {
    e.printStackTrace();
}
```

Особенностью блока **finally** можно считать то, код блока выполняется всегда, он выполняется даже в том случае, если перед ним были выполнены инструкции вида **return**, **break**, **continue**. Исключением из этого правила может быть разве что ситуация вызова **System.exit(0);** в **try** или **catch** блоке.

```
public class FinallyReturn {
    private static int number = 60;

    public static int getNumber() {
        try {
            return number / 2;
        } finally {
```

```
            return number * 2;
        }
    }

    public static void main(String[] args) {
        System.out.println(getNumber()); // out: 120
    }
}
```

7. Определение понятия исключения и исключительной ситуации. Оператор try-catch

Исключительная ситуация – это такая ситуация, которая произошла, но не была предусмотрена программой, то есть ситуация, в которой произошло отклонение от запланированного хода программы. **Исключительная ситуация** (**исключение**) – это проблема (ошибка), возникающая во время выполнения программы.

Исключение в Java – это **объект**, который описывает исключительную ситуацию, произошедшую в некоторой части кода. Когда исключительная ситуация возникает, создается объект, представляющий это исключение, этот объект выбрасывается в метод, который вызвал ошибку. Метод может обработать исключение внутри себя, либо передать это исключение дальше, где оно будет обработано. Исключения могут генерироваться исполнительной системой Java или можно генерировать их вручную в коде.

Обработка исключений в Java организована с помощью пяти ключевых слов:

- **try** – операции, которые нужно контролировать относительно генерации исключений, размещаются в блоке **try**. Если в блоке **try** происходит исключение, то оно является выброшенным этим блоком;
- **catch** – выброшенное исключение может быть перехвачено в блоке **catch** и обработано;
- **throw** – для ручного выброса исключений используется ключевое слово **throw**;
- **throws** – любое исключение кроме исключений **RuntimeException** и ошибок типа **Error**, которое выброшено из метода, следует определять с помощью ключевого слова throws в заголовке метода;
- **finally** – любой код, который обязательно должен быть выполнен при возврате из try-блока вне зависимости от того было сгенерировано исключение или нет, размещается в **finally**-блоке, который является заключительным блоком конструкции try-catch.

Общая форма try-catch:

```
try {
    // блок с операциями, которые нужно контролировать относительно генерации
исключений
} catch(ExceptionType1 ex) {
    // обработка исключений типа  ExceptionType1
} catch(ExceptionType2 ex) {
    // обработка исключений типа  ExceptionType2
} finally {
    // блок кода, который должен быть выполнен обязательно перед возвратом из блока try
}
```

Каждому блоку **try** должен соответствовать по крайней мере один блок **catch** или блок **finally**. Т.е. возможна ситуация, когда нет блоков **catch**, но есть блок **finally**, как возможна ситуация, когда есть блок(-и) **catch**, но нет блока **finally**. Исключение: **try-with-resources** (там при определенных обстоятельствах можно обойтись без **catch** и **finally**).

8. Ручная генерация исключений. Оператор throw

Программа может сама явно выбрасывать исключения, используя оператор **throw**. Это называется «ручным» выбросом исключения.

Общая форма использования оператора **throw**:

throw **ThrowableInstance**;

ThrowableInstance должен быть объектом типа **Throwable** или подкласса **Throwable**. Примитивные типы или классы, не являющиеся **Throwable** не могут использоваться в качестве исключений.

Получить объект **Throwable** можно двумя способами: использовать объект, полученный в блоке **catch** или создать **Throwable** объект с помощью оператора **new**.

При достижении оператора **throw** выполнение кода прекращается, блок **try/catch** проверяется на наличие соответствующего обработчика **catch**. Если он существует, то управление передается в этот блок **catch**. Создание объекта-исключения с помощью **new** без оператора **throw** не вызывает исключительную ситуацию.

Если метод может породить исключение, которое сам не обрабатывает, он должен передать это исключение вызывающему его методу, чтобы в нем оно было обработано или передано дальше. Передача исключений вызывающему методу в Java обеспечивается добавлением ключевого слова **throws** в заголовок объявления метода. После ключевого слова **throws** должны быть перечислены через запятую типы исключений, которые метод может выбросить, кроме исключений **RuntimeException** и ошибок **Error**.

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

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

Threads

1. Организация связи между потоками: назначение и работа с методами wait, notify.

Порой при взаимодействии потоков встает вопрос об извещении одних потоков о действиях других потоков. Например, в ситуации, когда действия одного потока зависят от результата действий другого потока, при этом нужно как-то известить первый поток о том, что второй поток произвел некие действия, а первый поток должен иметь возможность передать управление другому потоку, в результате действий которого он нуждается. Благодаря методам wait, notify, notifyAll() можно организовать эту связь между потоками. Эти методы реализованы как final-методы в классе Object, поэтому доступны всем классам.

- **final void wait() throws InterruptedException** – освобождает монитор и переходит в режим ожидания, пока другой поток, владеющий тем же монитором, не вызовет метод **notify()/ notifyAll()**;
- **final void wait(long timeout) throws InterruptedException** – аналогичен методу **wait()** с той разницей, что ожидает, что произойдет первым: вызов **notify()/notifyAll()** в другом потоке или закончится заданное время ожидания (**long timeout**). После того, как произойдет одно из этих событий, вне зависимости какое из них, поток выйдет из режима ожидания;
- **final void notify()** – «пробуждает» первый поток, который вызвал метод wait() на том же самом объекте;
- **final void notifyAll()** – «пробуждает» все потоки, которые вызывали wait() на том же самом объекте.

Рассматривая монитор и методы wait(), notify()/notifyAll() нужно сказать о следующем:
Существует два множества у объекта-монитора:
ENTRY SET – потоки, которые ожидают возможности войти в монитор;
WAIT SET – потоки, которые ожидают, что в другом потоке будет что-то сделано и они получат уведомление об этом;

Когда вызывается метод **wait()** у объекта-монитора:

1) поток снимает блокировку с монитора;
2) поток останавливается и переходит в состояние **WAITING**;
3) поток перемещается в **WAIT SET**;

Когда вызывается метод **notify()** у объекта-монитора:

1) берется первый поток по очередности из **WAIT SET** и перемещается в **ENTRY SET;**
2) поток из пункта 1 переходит в состояние **RUNNABLE**;
3) если при этом объект-монитор недоступен для входа, то поток переходит в состояние **BLOCKED**, т.е. блокируется в ожидании монитора;

Блокировка с объекта-монитора не снимается методами **notify()/notifyAll(),** она снимается только в том случае, если поток, владеющий монитором в данный момент, вышел из синхронизированного метода/блока, либо ушел в ожидание с помощью **wait()**.

2. Что такое «главный поток»? Как получить ссылку на главный поток выполнения программы?

При запуске программы автоматически создается и запускается один поток, который выполняет метод **main()**. Этот поток и называется главным. В главном потоке могут создаваться и запускаться другие потоки, которые в свою очередь тоже могут создавать и запускать потоки. Главный поток отличается от других потоков тем, что создается первым и порождает все другие потоки. Рекомендуется организовывать работу программы так, чтобы главный поток завершался последним, но это не является обязательным условием, дочерние потоки могут существовать и после завершения работы главного потока.

Несмотря на то, что главный поток создается автоматически, мы можем получить на него ссылку с помощью статического метода класса **Thread**:

- **static Thread currentThread()** – метод возвращает ссылку на поток, в котором был вызван.

Если метод **currentThread()** вызывается в методе **main()**, то будет возвращена ссылка на главный поток программы. Получив такую ссылку, мы получаем возможность манипулировать главным потоком точно так же, как и любым другим.

3. Методы isAlive() и getState() класса Thread

Может возникнуть такая ситуация, при которой необходимо узнать состояние потока. Например, закончил ли поток свое действие или нет.

Существует два метода для определения состояния потока:

- **final boolean isAlive()** – возвращает true, если поток, для которого был вызван этот метод, все еще выполняется. В противном случае возвращается false;
- **Thread.State getState()** – возвращает одну из констант перечисления Thread.State, определяющую состояние потока

Состояния, которые может вернуть getState():

- **NEW** – поток создан, но еще не запущен; isAlive() = false
- **RUNNABLE** – поток выполняется; isAlive() = true
- **BLOCKED** – поток блокирован; isAlive() = true
- **WAITING** – поток ожидает окончания работы другого потока; isAlive() = true
- **TIMED_WAITING** – поток ждет некоторое время окончания работы другого потока; isAlive() = true
- **TERMINATED** – поток завершен; isAlive() = false

4. Потоки демоны: назначение, создание, случаи применения

Потоки-демоны – это такие потоки, которые работают в фоновом режиме вместе с программой, но не являются неотъемлемой частью логики программы. Если есть какой-то процесс, который нужен для обслуживания основных потоков программы, и он может выполняться на фоне работы основных потоков, то деятельность такого процесса может быть запущена в виде потока-демона. Все потоки, которые порождаются потоками-демонами, являются потоками-демонами.

С помощью метода **setDaemon(boolean value)** мы можем до запуска потока указать, что поток является **потоком-демоном**. Метод **boolean isDaemon()** позволяет определить, является ли поток демоном или нет.

Традиционно принято считать, что **программа Java** завершает работу, когда завершают работу все ее потоки. На самом деле это не совсем так. Кроме созданных нами потоков существуют системные потоки: поток работы сборщика мусора, другие служебные потоки, созданные **JVM**. Такие потоки остановить мы не можем. Выходит, что **программа Java** не может завершить свою работу, ведь существуют потоки, которые мы не можем остановить. Но все-таки она ее завершает. Все эти системные потоки являются **потоками-демонами**, а **программа Java** на самом деле завершает работу, если завершили свою работу все потоки, не являющиеся **демонами**. Т.к. потоки-демоны созданы для обслуживания основных потоков, существование их без основных потоков лишено смысла, поэтому они завершают свою работу, когда все потоки не-демоны завершили ее.

5. Что такое синхронизация, понятие монитора

Многопоточность позволяет обеспечить асинхронное поведение программы. При этом периодически возникает необходимость правильной синхронизации работы потоков.

Например, требуется, чтобы два потока совместно взаимодействовали, используя некую структуру данных, например, список. В таком случае нужно гарантировать отсутствие конфликтов между потоками, иначе результаты их работы могут быть непредсказуемы.

Если несколько потоков нуждаются в доступе к одному ресурсу, необходимо гарантировать такое поведение программы, при котором ресурс будет использоваться одновременно только одним потоком. Процесс, который позволяет это гарантировать, называется **синхронизацией**.

Монитор – это объект, который используется для взаимоисключающей блокировки (mutex).

Одновременно только один поток может **захватить** и **держать** монитор. Когда поток **захватывает** монитор, он **входит** в монитор, а монитор **блокируется** для входа. Все остальные потоки, пытающиеся **войти** в блокированный монитор, будут приостановлены в ожидании **разблокировки** монитора, т.е. другие потоки **ожидают** монитор.

Базовая синхронизация с монитором в Java обеспечивается с помощью ключевого слова **synchronized**. Синхронизация может осуществляться на уровне:

- синхронизированных методов;
- синхронизированных блоков;

Уровень синхронизованных методов

В Java каждый объект имеет свой собственный неявный связанный с ним монитор. Чтобы ввести монитор объекта, вызывают метод, в сигнатуре которого фигурирует ключевое слово **synchronized**. Пока поток находится внутри синхронизированного метода, другие потоки, пытающиеся вызвать его или другие синхронизированные методы этого же объекта, будут ожидать освобождения монитора объекта. Для выхода из монитора и его разблокировки, поток, владеющий монитором, просто возвращается из синхронизированного метода.

Добавление в сигнатуру метода ключевого слова **synchronized** означает, что поток должен получить блокировку монитора (захватить монитор) объекта перед входом в этот метод. Если же метод является **статическим**, тогда это означает необходимость захвата блокировки, относящейся к объекту **Class**.

Уровень синхронизованных блоков

Если необходимо синхронизировать доступ к объектам класса, который не был разработан для многопоточного доступа, т.е. в этом классе нет методов с ключевым словом **synchronized**, то для синхронизации можно применить синхронизированный блок.

Общая форма блока синхронизации:

```
synchronized(lock) {
        // операторы для синхронизации
}
```

lock – ссылка на объект, который нужно синхронизировать. Синхронизированный блок гарантирует, что вызов метода у объекта *lock*, происходит только после того, как текущий поток захватит монитор объекта *lock.*

6. Опишите жизненный цикл потока

При выполнении программы объект класса Thread может находиться в одном из состояний, соответствующих элементам вложенного в класс **Thread** перечисления **Thread.State**:

- **NEW** – поток создан, но еще не запущен;
- **RUNNABLE** – поток выполняется;
- **BLOCKED** – поток блокирован;
- **WAITING** – поток ожидает окончания работы другого потока;
- **TIMED_WAITING** – поток ждет некоторое время окончания работы другого потока;
- **TERMINATED** – поток завершен;

Получить текущее состояние потока можно с помощью вызова метода **getState()**.

При создании потока он получает состояние **NEW** и не выполняется. Для смены состояния **NEW** на состояние **RUNNABLE** следует выполнить метод **start()**, который вызовет метод **run()** – основной метод потока.

Поток может перейти в состояние ожидания **WAITING** при вызове методов **join()**, **wait()**, **suspend()** (deprecated). Для задержки потока на некоторое время можно перевести его в режим ожидания **TIMED_WAITING** с помощью методов **sleep(long millis)**, **wait(long timeout)**, **join(long timeout)**.

Поток может получить обратно состояние **RUNNABLE** либо с помощью вызова метода **resume()** (deprecated), если поток попал в **WAITING** состояние после вызова метода **suspend()** (deprecated), либо с помощью вызова метода **notify()/notifyAll()**, если поток попал в **WAITING** состояние после вызова метода **wait()**.

Поток переходит в состояние **TERMINATED**, если вызваны методы **interrupt()**, **stop()** (deprecated) или метод **run()** завершил свое выполнение. После получения этого статуса запустить поток повторно уже нельзя.

Состояние **BLOCKED** получают все потоки, которые ожидали монитор в состоянии **WAITING**, получили уведомление от другого потока после вызова **notify()/notifyAll()**, стали **RUNNABLE**, но не были выбраны планировщиком **JVM** для получения монитора (либо монитор еще не освободился), т.к. только один из потоков может владеть монитором, остальные потоки, готовые к работе и желающие войти в этот монитор, переходят в состояние **BLOCKED** – состояние ожидания монитора.

7. Классы синхронизированных коллекций пакета java.util.concurrent

В **Java SE 5** был добавлен пакет **java.util.concurrent**, классы которого обеспечивают более высокую производительность, масштабируемость, построение потокобезопасных блоков concurrent классов, вызов утилит синхронизации, в него добавлены классы семафоров и блокировок.

Обзор java.util.concurrent (7)

- параллельные аналоги существующих классов-коллекций **ConurrentHashMap** (эффективный аналог **Hashtable**), **ArrayBlockingQueue** (**FIFO** очередь с фиксированой длиной), **PriorityBlockingQueue** (очередь с приоритетом) и **ConcurrentLinkedQueue** (**FIFO** очередь с нефиксированой длиной);
- классы **CopyOnWriteArrayList** и **CopyOnWriteArraySet**, копирующие свое содержимое при попытке его изменения, причем ранее полученные итераторы будут корректно продолжать работать с исходным набором данных;
- блокирующие очереди **BlockingQueue** и **BlockingDeque**, гарантирующие остановку потока, запрашивающего элемент из пустой очереди, до появления в ней элемента, доступного для извлечения. Эти очереди также блокируют поток, пытающийся вставить элемент в заполненную очередь, до тех пор, пока в очереди не освободится место под этот элемент;
- механизм управления заданиями, основанный на возможностях класса **Executor**, включающий организацию запуска пула потоков и службы их планирования;
- классы-барьеры синхронизации общего назначения, такие как **Semaphore** (предлагает потоку ожидать завершения действий в других потоках), **CyclicBarrier** (предлагает нескольким потокам ожидать момента, когда они все достигнут какой-либо точки, после чего барьер снимается);
- класс **Exchanger** позволяет потокам синхронизироваться и обмениваться информацией;
- классы атомарных переменных (**AtomicInteger**, **AtomicLong**, **AtomicReference**), а также их более производительные аналоги **SynchronizedInt** и т.п.;

8. Метод join() класса Thread

Метод **join()** – ждет завершения потока, для которого он был вызван, чтобы только после этого перейти к дальнейшему исполнению кода.

Основная форма метода **join()**:

- **final void join() throws InterruptedException**

Дополнительные формы метода **join()** позволяют определить максимальное время ожидания завершения указанного потока, если по истечении этого времени поток не завершит работу, код следующий за вызовом этого метода начнет исполняться:

- **final void join(long millis) throws InterruptedException**
- **final void join(long millis, int nanos) throws InterruptedException**

```
public static void example() throws InterruptedException {
    Thread thread = new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 10; ++i) {
                System.err.println(i);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    });
```

```
        System.err.println("start... ");
        thread.start();
        thread.join();
        System.err.println("finish.");
    }
```

Java Language

1. Пакеты (package): правила создания, правила именования, назначение

Пакет (package) – коллекция классов, объединенная в логическую группу. **Пакеты** упрощают добавление классов в исходный код и позволяют избегать конфликта имен между классами. Имена классов в одном пакете никак не пересекаются с именами классов в другом пакете, т.к. полные имена классов включают наименование пакета, в котором они лежат.

Любой класс относится к какому-то пакету, который может быть неименованным (default package). Если программный файл находится в неименованном пакете, оператор *package* отсутствует в нем. Оператор *package*, помещаемый в начале исходного кода программного файла, определяет именованный пакет.

package people; // первая инструкция в программном файле

Пакет тесно связан со структурой каталогов в файловой системе. Все классы пакета *people* должны лежать в соответствующем каталоге *people*.

При использовании классов из других пакетов, необходимо перед именем класса добавлять полное имя пакета через точку, т.е. использовать полное имя класса. Но можно импортировать классы из других пакетов или импортировать пакеты целиком:

import people.Student; // импорт класса Student из пакета people

import people.; // импорт всего пакета*

Нельзя импортировать 2 класса с одинаковым именем, но можно импортировать один из них, а для второго пользоваться полным именем класса.

Основные правила работы с пакетами:

- оператор **package** в исходном коде программного файла может быть только один и конструкция с этим оператором должна быть первой;
- оператор **package** в исходном коде программного файла может отсутствовать;
- операторов **import** может быть в исходном коде программного файла сколько угодно;
- имя пакета должно совпадать с именем соответствующего каталога в файловой системе;

Основные правила именования пакетов:

- обратный интернет-адрес производителя или заказчика ПО: для заказчика www.vk.com это будет com.vk;
- далее следует имя проекта, например, profile;
- затем располагаются пакеты, определяющие приложение, например, music;
- итоговый пакет music полностью выглядит так: com.vk.profile.music;

2. Статические методы, особенности работы со статическими методами

Статические методы – методы, объявленные в классе как static, являются общими для всех объектов класса и называются методами уровня класса. Статические методы не привязаны ни какому объекту и не содержат указателя **this** на конкретный экземпляр класса, вызвавший метод. Статические методы реализуют парадигму «**раннего связывания**», определяющую версию метода на этапе компиляции.

По причине недоступности **this** статические методы не могут обращаться к нестатическим полям и методам напрямую, т.к. они не знают, к какому объекту они относятся, да и сам экземпляр класса может не быть создан к тому моменту, когда будет вызван статически метод. Для обращения к статическим методам достаточно имени класса, в котором они определены.

Вызов статического метода следует осуществлять с помощью указания имени класса, а не объекта. Несмотря на то, что статический метод можно вызвать с использованием имени объекта и такой вызов не повлечет появление ошибок, вызов статического метода через имя объекта не является логически корректным и снижает качество кода.

Переопределение статических методов **невозможно**, так как обращение к статическому методу осуществляется посредством имени класса, к которому они принадлежат.

Нестатические методы могут обращаться к статическим методам без каких-либо дополнительных усилий.

3. Анонимные классы: объявление и правила работы

Анонимные (anonymous) классы применяются для придания уникальной функциональности отдельно взятому экземпляру класса, например, для обработки событий, реализации блоков прослушивания, реализации интерфейсов, запуска потоков и т.д.

Анонимные классы эффективно используются для переопределения нескольких методов и создания собственных методов и полей объекта. Этот прием эффективен в случае, когда необходимо переопределение метода, но создавать новый класс нет необходимости из-за узкой области или одноразового применения метода.

Анонимный класс:

- расширяет другой класс или реализует интерфейс при определении одного единственного объекта, остальным объектам будет соответствовать реализация, определенная в самом классе;
- объявление анонимного класса выполняется одновременно с созданием его объекта с помощью оператора new;
- конструкторы анонимных классов ни определить, ни переопределить нельзя;
- анонимные классы допускают вложенность друг в друга;

Пример:

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 10; ++i) {
            System.err.println(i);
        }
    }
});
```

Анонимным классом в данном примере выступает безымянный класс, который передается в конструктор потока *new Thread()*, реализующий интерфейс *Runnable*, переопределяющий метод *run()*;

4. Вложенные классы: объявление и правила работы

В Java можно объявлять классы внутри классов и даже внутри методов. Эти классы делятся на **внутренние (inner) нестатические**, **вложенные (nested) статические** и **анонимные (anonymous) классы**.

Нестатические внутренние классы (8):

- доступ к элементам внутреннего класса возможен только из внешнего класса через объект внутреннего класса;
- методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса;
- объект внутреннего класса имеет ссылку на объект своего внешнего класса;
- внутренний класс не может содержать статических полей и методов кроме static final;
- внутренний класс может быть суперклассом, производным, реализующим интерфейсы;
- внутренний класс может быть объявлен как final, abstract, private, protected, public;
- если необходимо создать объект внутреннего класса где-то кроме нестатического метода внешнего класса, то нужно определить тип объекта как имя_внешнего_класса.имя_внутреннего_класса;
- внутренний класс может быть объявлен внутри метода или логического блока внешнего класса, видимость класса регулируется видимостью того блока, в котором он объявлен. Однако внутренний класс сохраняет доступ ко всем полям и методам внешнего класса, а также к константам, объявленным в текущем блоке кода.

Статические вложенные классы (6):

- класс, вложенный в интерфейс, статический по умолчанию;
- вложенный класс может быть суперклассом, производным, реализующим интерфейсы;
- статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса;
- вложенный класс имеет доступ к статическим полям и методам внешнего класса;

- подкласс вложенного класса не наследует возможность доступа к членам внешнего класса, которым наделен его суперкласс;
- статический метод вложенного класса вызывается при указании полного относительного пути к нему.

Анонимный класс (4):

- расширяет другой класс или реализует интерфейс при определении одного единственного объекта, остальным объектам будет соответствовать реализация, определенная в самом классе;
- объявление анонимного класса выполняется одновременно с созданием его объекта с помощью оператора new;
- конструкторы анонимных классов ни определить, ни переопределить нельзя;
- анонимные классы допускают вложенность друг в друга.

При выборе между inner, nested и anonymous классами следует учитывать их особенности:

- **INNER**. При необходимости жесткой связи между объектами внутреннего и внешнего классов следует реализовывать внутренний класс. Благодаря доступу к любым полям и методам внешнего класса и возможности быть подклассом других классов внутренние классы позволяют решить проблемы множественного наследования, когда требуется наследовать свойства нескольких классов;
- **NESTED**. Если не существует жесткой необходимости связи объекта внутреннего класса с объектом внешнего класса, то есть смысл делать такой класс статическим. Вложенный класс логически связан с классом-владельцем, но может быть использован независимо от него;
- **ANONYMOUS**. Анонимные классы эффективны в случае, когда необходимо переопределение метода, но создавать новый класс нет необходимости из-за узкой области или одноразового применения метода.

5. Интерфейсы в Java 7: объявление, методы и поля интерфейсов

Интерфейсы в Java 7

Интерфейс описывает функциональность, которую должен реализовать каждый класс, который его реализует. **Интерфейсы** подобны абстрактным классам, хотя и не являются классами. Все объявленные в интерфейсе **методы** автоматически трактуются как **public** и **abstract**, а все **поля** – как **public, static, final**, даже если какие-то из ключевых слов не присутствуют в объявлении методов и полей. Объявленные методы интерфейса не могут быть реализованы в нем самом.

В Java существуют два вида интерфейсов:

- интерфейсы, определяющие функциональность для классов посредством описания методов;
- интерфейсы, реализация которых автоматически придает классу определенные свойства (**tagged**-интерфейсы). Например, Cloneable, Serializable;

Интерфейс может быть наследником нескольких интерфейсов. Классы же интерфейсы только реализуют. Класс может наследовать только один суперкласс и реализовывать любое число интерфейсов, указываемых через запятую после ключевого слова **implements**, дополняющего определение класса. После этого в классе должны быть реализованы все методы интерфейса, в противном случае класс необходимо объявить абстрактным.

Общее определение интерфейса:

```
[public] interface Name [extends Name1, Name2, ... NameN] {
        // реализация интерфейса
}
```

Реализация интерфейса:

```
[public] class Name [extends ClassName] [implements Name1, Name2, ... NameN] {
        // реализация класса
}
```

Нельзя создать экземпляр интерфейса, но можно объявлять **ссылки интерфейсного типа**. Интерфейсная ссылка может указывать на экземпляр любого класса, который реализует интерфейс того же типа, что и сама ссылка. При вызове метода через такую ссылку будет

вызываться его реализованная версия у объекта, на который указывает эта ссылка. Вызываемый метод определяется с помощью динамического связывания во время выполнения.

6. Перегрузка методов: правила перегрузки, разрешение перегрузки

Перегрузка методов (**overloading**) – определение методов с одинаковым наименованием, но различной сигнатурой. Фактически такие методы являются разными методами с совпадающим наименованием. Сигнатура метода определяется наименованием метода, числом и типом параметров.

Перегрузка реализует механизм «**раннего связывания**», то есть версия вызываемого метода определяется на этапе компиляции. **Перегрузка** может ограничиваться одним классом, а может не ограничиваться, если методы с одинаковым именем и различным списком параметров находятся в разных классах одной цепочки наследования.

Если в суперклассе и в подклассе определены методы с идентичной сигнатурой, то данное определение методов является не их **перегрузкой** (**overloading**), а **переопределением** (**overriding**), то есть совершенно другим механизмом, который вдобавок реализует парадигму «позднего связывания». Не следует путать **перегрузку** методов и их **переопределение**.

Статические методы могут перегружаться нестатическими и наоборот – без ограничений.

При передаче объекта в метод выбор метода производится в зависимости от **типа ссылки** на этапе компиляции.

При перегрузке методов желательно придерживаться следующих правил:

- стараться не использовать сложные варианты перегрузки;
- стараться избегать использования перегрузки с одинаковым числом параметров;
- стараться заменять (по возможности) перегруженные методы на несколько разных методов;

Конструкторы классов тоже могут перегружаться.

7. Абстрактные классы и абстрактные методы

Абстрактные классы отражают некие абстрактные общие модели для целого ряда сущностей.

Абстрактные классы объявляются с ключевым словом **abstract** и содержат объявления абстрактных методов, которые не реализованы в этих классах, а их реализация подразумевается в подклассах. Объекты таких классов нельзя создать, но можно создавать объекты подклассов, которые реализуют все абстрактные методы суперкласса. Ссылки абстрактного типа в программе могут использоваться, но ссылаться они могут только на объекты подклассов, которые содержат конкретную реализацию всех абстрактных методов.

Абстрактные классы могут содержать полностью реализованные методы, конструкторы, поля данных. На самом деле абстрактный класс может даже не содержать ни одного абстрактного метода, но объявление его с ключевым словом **abstract** говорит о том, что экземпляры такого класса создавать нельзя, и он является абстрактным.

В общем случае абстрактный класс объявляет требования к функциональности, для своих подклассов. Например, абстрактный класс **Animal** с абстрактным методом **say()** объявляет требование, по которому все подклассы должны в соответствии со своими особенностями реализовать метод **say()**.

Подкласс абстрактного класса должен реализовать все абстрактные методы своего суперкласса. Если какой-то из абстрактных методов подкласс не реализует, то такой подкласс должен быть тоже объявлен с ключевым словом **abstract**.

Абстрактные методы размещаются в абстрактных классах или интерфейсах, тела таких методов отсутствуют, подразумевается, что такие методы должны быть реализованы в подклассах абстрактного класса или в классах, реализующих интерфейс.

Пример абстрактного метода:

public abstract void run();

Спецификатор **abstract** присутствует как в объявлении методов абстрактного класса, так и в объявлении самого абстрактного класса.

В интерфейсах для методов спецификатор **abstract** можно не указывать. Если они не являются default-методами или static-методами (Java 8), то они по умолчанию будут абстрактными.

8. Статические поля, статические константные поля

Статические поля – поля, объявленные в классе как **static**, являются общими для всех объектов класса и называются переменными класса. Если один объект изменит значение такого поля, то изменение увидят все объекты этого класса. И статические, и нестатические методы могут обращаться к статическим полям напрямую без дополнительных усилий. Для обращения к статическим полям достаточно имени класса, в котором они определены.

Константные поля – это поля, объявленные в классе как **final**. **Константное поле** должно быть проинициализировано либо при объявлении, либо в конструкторе, либо в логическом блоке. Значение по умолчанию константное поле не получает в отличие от других переменных класса. **Константные поля** являются неизменными полями класса, инициализируются один раз, и им нельзя присвоить другое значение. Нестатические константные поля являются **константами уровня объекта**, то есть у каждого объекта этого класса будет своя **константа**.

Статические константные поля – статические поля класса, объявленные как **final**. Такие константные поля являются **константами уровня класса**, к ним можно обращаться с помощью имени класса. **Статические константные поля** должны быть проинициализированы при объявлении. Имена **static final** переменных принято записывать в верхнем регистре с символом подчеркивания между словами.

Пример:

static final String OUTPUT_ERROR = “Output error!”;

Константы ссылочного типа не делают объект неизменяемым. Константы ссылочного типа – это константные ссылки, которые подразумевают, что данная переменная ссылочного типа не может быть повторно проинициализирована и указывать на какой-то другой объект, но сам объект может изменяться, если это позволяют его методы.

Например:

```
final List<String> list = new ArrayList<>();
list.add(“something”);      // ок
list = new ArrayList<>(); // ошибка компиляции
```

9. Интерфейсы в Java 8

Интерфейсы в Java 7

Интерфейс описывает функциональность, которую должен реализовать каждый класс, который его реализует. **Интерфейсы** подобны абстрактным классам, хотя и не являются классами. Все объявленные в интерфейсе **методы** автоматически трактуются как **public** и **abstract**, а все **поля** – как **public, static, final**, даже если какие-то из ключевых слов не присутствуют в объявлении методов и полей. Объявленные методы интерфейса не могут быть реализованы в нем самом.

В Java существуют два вида интерфейсов:

- интерфейсы, определяющие функциональность для классов посредством описания методов;
- интерфейсы, реализация которых автоматически придает классу определенные свойства (**tagged**-интерфейсы). Например, Cloneable, Serializable;

Интерфейс может быть наследником нескольких интерфейсов. Классы же интерфейсы только реализуют. Класс может наследовать только один суперкласс и реализовывать любое число интерфейсов, указываемых через запятую после ключевого слова **implements**, дополняющего определение класса. После этого в классе должны быть реализованы все методы интерфейса, в противном случае класс необходимо объявить абстрактным.

Общее определение интерфейса:

```
[public] interface Name [extends Name1, Name2, ... NameN] {
        // реализация интерфейса
}
```

Реализация интерфейса:

```
[public] class Name [extends ClassName] [implements Name1, Name2, ... NameN] {
        // реализация класса
}
```

Нельзя создать экземпляр интерфейса, но можно объявлять **ссылки интерфейсного типа**. Интерфейсная ссылка может указывать на экземпляр любого класса, который реализует интерфейс того же типа, что и сама ссылка. При вызове метода через такую ссылку будет вызываться его реализованная версия у объекта, на который указывает эта ссылка. Вызываемый метод определяется с помощью динамического связывания во время выполнения.

Интерфейсы в Java 8

Начиная с Java 8 мы можем использовать в интерфейсах методы по умолчанию (**default methods**) и статические методы (**static methods**).

Java 8 позволяет добавлять неабстрактные реализации методов в интерфейс, используя ключевое слово **default** в объявлении метода. Классы, реализующие интерфейс с **default**-методом, должны переопределять только абстрактные методы, переопределить **default**-методы можно, но в этом нет необходимости. Таким образом, класс может просто пользоваться уже реализованным в интерфейсе **default**-методом.

Если какой-либо класс в иерархии наследования уже имеет метод с той же сигнатурой, что и default-метод, то метод по умолчанию становятся неактуальным. default-методы нужно использовать с осторожностью, чтобы не столкнуться с ромбовидной проблемой множественного наследования.

В Java 8 можно добавлять неабстрактные реализации статических методов в интерфейс, используя ключевое слово **static** в объявлении метода.

Статические методы в интерфейсе **(4)**:

- являются частью интерфейса, нельзя их использовать для объектов класса реализации;
- не могут быть переопределены, если в классе, реализующем этот интерфейс есть метод с идентичной сигнатурой, он не будет переопределять метод интерфейса, а будет являться самостоятельным методом класса;
- могут быть полезны для обеспечения вспомогательных методов для совершения каких-то проверок, сортировок и т.п.
- удобны для создания вспомогательных интерфейсов, состоящих из статических функций. Это может быть хорошей заменой вспомогательным классам.

Функциональные интерфейсы – еще одно нововведение в Java 8. Интерфейс с одним абстрактным методом является функциональным, для обозначения такого интерфейса можно использовать введенную в Java 8 аннотацию **@FunctionalInterface**. Эта аннотация не является обязательной, но она очень удобна, т.к. предупреждает случайные добавления абстрактных методов в функциональный интерфейс. Функциональные интерфейсы могут содержать сколько угодно **default**-методов или **static**-методов, но абстрактный метод должен быть только один. Функциональные интерфейсы позволяют использовать лямбда-выражения для создания анонимных классов, реализующих данный интерфейс.

Computer Science

1. **Что такое сортировка, зачем она нужна, какие виды сортировок бывают, какие у них свойства**

Сортировка – это алгоритм упорядочивания элементов по какому-либо признаку. Сортировка необходима для обеспечения удобства поиска информации. Например, у нас есть список из 1000 людей разного возраста. В данный момент нам может быть необходимо найти 20 самых младших людей, можно проходить каждый раз по всему массиву данных о людях, находить самого младшего, записывать в отдельную структуру данных, потом проходить вновь, искать следующего младшего, проверяя, что такого уже не записано в нашу результирующую структуру данных. Это не очень удобно. В отсортированных данных по возрасту нам нужно было бы просто отобрать первые 20 (или последние – в случае убывающей сортировки) человек и все. К тому же это в данный момент нам нужно было отобрать 20 самых младших людей, а спустя пять минут будет нужно найти 4 старших человека или людей определенного возраста. Поиск информации в беспорядочно расположенных данных трудоемок. В отсортированных данных поиск информации происходит гораздо быстрее, тем более для отсортированных данных могут применяться специальные виды поиска (например, бинарный поиск), которые ищут информацию с еще большей производительностью.

Основные свойства сортировок:

- **время работы** – это свойство зачастую считается наиболее важным. Оценивается худшее, среднее и лучшее время работы алгоритма. У большинства алгоритмов временные оценки бывают O(n*log n), O(n^2);
- **дополнительная память** – свойство сортировки, показывающее, сколько дополнительной памяти требуется алгоритму. Сюда входят дополнительный массив, переменные, затраты на стек вызовов. Обычно доп. затраты памяти составляют O(1), O(log n), O(n);
- **устойчивость** – устойчивой называется сортировка, не меняющая порядок объектов с одинаковыми ключами. Ключ – поле элемента, по которому проводим сортировку;
- **количество обменов** – этот параметр может быть важен в том случае, если обмениваемые объекты имеют большой размер, т.к. при большом количестве обменов время работы сильно увеличивается;

Некоторые виды сортировок:

- **Пузырьковая сортировка**
  - время работы: лучшее – O(n), среднее – O(n^2), худшее – O(n^2);
  - затраты памяти – O(1);
  - устойчивая;
  - количество обменов – обычно O(n^2) обменов;
- **Сортировка выбором**
  - 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;
- **Сортировка вставками**
  - время работы: лучшее – O(n), среднее – O(n^2), худшее – O(n^2);
  - затраты памяти – O(1);
  - устойчивая;
  - количество обменов – обычно O(n^2) обменов;
- **Быстрая сортировка**
  - время работы: лучшее – O(n*log n), среднее – O(n*log n), худшее – O(n^2);
  - затраты памяти – O(log n);
  - неустойчивая;
  - количество обменов – обычно O(n*log n) обменов;
- **Сортировка слиянием**
  - время работы: лучшее – O(n*log n), среднее – O(n*log n), худшее – O(n*log n);
  - затраты памяти – O(n);
  - устойчивая;

- количество обменов – обычно O(n*log n) обменов;

- **Поразрядная сортировка**
  - время работы: лучшее – O(nk), среднее – O(nk), худшее – O(nk);
  - затраты памяти – O(n + k);
  - устойчивая;
  - количество обменов – обычно O(nk) обменов;

2. Бинарный поиск, принцип и краткое описание алгоритма

Ответ дан с использованием термина «множество» для упрощения пояснения, поиск же может применяться и к другим структурам данных.

Бинарный поиск – алгоритм поиска объекта по заданному признаку в множестве объектов, упорядоченных по тому же признаку.

Принцип. На каждом шаге множество объектов делится на 2 части и в работе остается та часть, где находится искомый объект, деление множества на 2 части продолжается до тех пор, пока элемент не будет найден, либо пока не будет установлено, что его нет в заданном множестве.

Алгоритм бинарного поиска (на отсортированной по возрастанию структуре данных):

1) задаем границы: левая граница = 0вой индекс; правая граница = максимальный индекс структуры данных;
2) берем элемент посередине между границами, сравниваем его с искомым;
3) оцениваем результат сравнения:
   - если искомое равно элементу сравнения, возвращаем индекс элемента, на этом работа алгоритма заканчивается;
   - если искомое больше элемента сравнения, то сужаем область поиска таким образом: левая граница = индекс элемента сравнения + 1;
   - если искомое меньше элемента сравнения, то сужаем область поиска таким образом: правая граница = индекс элемента сравнения - 1;
4) Повторяем шаги 1-3 до тех пор, пока правая граница не станет меньше левой;
5) Возвращаем -1 (до этого шага доходим только в случае, если элемент не был найден);

Время выполнения алгоритма – O(log n);

3. Quick Sort, принцип и краткое описание алгоритма

Ответ дан с использованием термина «массив» для упрощения пояснения, сортировка же может применяться и к другим структурам данных.

Принцип

Опирается на принцип «разделяй и властвуй». Выбирается опорный элемент, это может быть любой элемент. От выбора элемента не зависит корректность работы алгоритма, но в отдельных случаях выбор этого элемента может повысить эффективность работы алгоритма. Оставшиеся элементы сравниваются с опорным и переставляются так, чтобы массив представлял собой последовательность: элементы меньше опорного-равные опорному элементы-элементы больше опорного. Для частей «больше» и «меньше» опорного элемента рекурсивно выполняется та же последовательность операций, если размер этой части составляет больше 1 элемента.

На практике входные данные обычно делят не на 3, а на 2 части. Например, «меньше опорного элемента» и «больше или равны опорному элементу». В общем случае разделение на 2 части эффективнее.

Алгоритм

1) проверяется, что во входном массиве больше 1 элемента, в противном случае алгоритм завершает свое действие;
2) с помощью какой-то схемы разбиения (например, схема разбиения Хоара) элементы в массиве меняются местами и определяется опорный элемент;
3) массив разбивается на 2: «меньше опорного элемента» и «больше или равны опорному элементу»;
4) рекурсивно вызывается этот же алгоритм для частей массива, полученных в пункте 3;

Разбиение Хоара

Эта схема разбиения подразумевает использование 2 индексов (с начала массива и с конца массива), которые приближаются навстречу друг к другу, пока найдется пара элементов, где один элемент больше опорного и расположен перед ним, а другой меньше опорного и расположен

после него. Эти элементы меняются местами. Поиск таких пар элементов и их обмен происходит до тех пор, пока индексы не пересекутся. Алгоритм возвращает последний индекс. Эта схема разбиения является одной из наиболее эффективных.

Свойства

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

Плюсы

- в среднем очень быстрая;
- требует небольшое количество дополнительной памяти;

Минусы

- зависит от данных, на плохих данных деградирует до O(n^2);
- может привести к ошибке переполнения стека;
- неустойчива;

4. Merge Sort, принцип и краткое описание алгоритма

Ответ дан с использованием термина «массив» для упрощения пояснения, сортировка же может применяться и к другим структурам данных.

Принцип

Основная идея заключается в том, чтобы разбить массив на максимально малые части, которые могут считаться отсортированными (массивы с одним элементом), а потом сливать их между собой в порядке, необходимом для сортировки.

Алгоритм

1) Если в массиве меньше 2 элементов, то он уже отсортирован, алгоритм завершает свою работу;
2) Массив разбивается на 2 части (примерно пополам), для которых рекурсивно вызывается тот же алгоритм сортировки;
3) После сортировки двух частей массива производится слияние двух упорядоченных массивов в один упорядоченный массив;

Процедура слияния:

1) объявляются счетчики индексов для результирующего массива и для двух сливаемых частей, счетчики инициализируются 0;
2) в цикле с условием (пока не дошли до конца какого-либо из сливаемых массивов) на каждом шаге берется меньший из двух первых (по индексу счетчиков) элементов сливаемых массивов и записывается в результирующий массив. Счетчики индексов результирующего массива и массива, из которого был взят элемент, увеличиваются на 1.
3) все оставшиеся элементы сливаемого массива, счетчик которого не дошел до конца массива, записываются в результирующий массив.

Свойства

- время работы: лучшее – O(n*log n), среднее – O(n*log n), худшее – O(n*log n);
- затраты памяти – O(n);
- устойчивая;
- количество обменов – обычно O(n*log n) обменов;

Плюсы

- проста для понимания;
- независимо от входных данных стабильное время работы;
- устойчивая;

Минусы

- нужно O(n) дополнительной памяти;
- в среднем на практике несколько уступает в скорости Quick Sort;

5. Insertion Sort, принцип и краткое описание алгоритма

Ответ дан с использованием термина «массив» для упрощения пояснения, сортировка же может применяться и к другим структурам данных.

Принцип

Есть часть массива (в роли этой части может выступать первый элемент в массиве), которая уже отсортирована, требуется вставить остальные элементы в отсортированную часть, сохранив упорядоченность.

Алгоритм

1) выбирается один из элементов входных данных и вставляется на нужную позицию в уже отсортированной части массива;
2) шаг 1 повторяется до тех пор, пока весь массив не будет отсортирован;

Обычно в качестве отсортированной части массива изначально выступает первый элемент (элемент с индексом 0), порядок выбора очередного элемента для вставки в отсортированную часть произволен, но обычно с целью достижения устойчивости алгоритма элементы вставляются по порядку их появления в исходном массиве.

Свойства

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

Плюсы

- проста в реализации;
- не нужна дополнительная память;
- ускоряется на частично отсортированных массивах;
- хороша на малом количестве элементов (<100);
- устойчивая;

Минусы

- медленная в среднем и худшем случае;

6. Selection Sort, принцип и краткое описание алгоритма

Ответ дан с использованием термина «массив» для упрощения пояснения, сортировка же может применяться и к другим структурам данных.

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. Radix Sort, принцип и краткое описание алгоритма

Ответ дан с использованием термина «массив» для упрощения пояснения, сортировка же может применяться и к другим структурам данных.

Поразрядная сортировка может быть двух типов LSD (least significant digit) и MSD (most significant digit): отличие в том, с какого разряда начинается анализ, т.е. с меньшего или большего соответственно. В ответе рассматриваю LSD версию поразрядной сортировки.

Принцип

Алгоритм представляет собой цикл по номеру разряда от младшего к старшему. На каждой итерации в результирующем массиве числа упорядочиваются по разряду с помощью какой-либо устойчивой сортировки. Чаще всего в качестве вспомогательной сортировки используется сортировка подсчетом (Counting Sort).

Алгоритм

1) Находится максимальный элемент среди элементов массива для определения количества разрядов;
2) В цикле по номеру разряда сортируется результирующий массив по разряду с помощью сортировки подсчетом;

Сортировка подсчетом (Counting Sort)

Обозначения: входной массив **ar**, вспомогательный массив **count** для счетчика, массив **res** для результата.

1) заполняем массив count 0ми;
2) для каждого ar[i] увеличиваем count[a[i]] на 1;
3) подсчитываем количество элементов меньше или равных j-ому индексу для каждого элемента. Для этого каждый count[j], начиная с j = 1, увеличиваем на count[j - 1]. Таким образом, в последней ячейке будет находиться количество элементов равное количеству элементов в массиве.
4) В цикле (по всем элементам входного массива) входной массив читается с конца, значение count[ar[i]] уменьшается на 1 и в res[count[ar[i]]] записывается a[i]. Проход по массиву с конца обеспечивает устойчивость сортировки.

Свойства

- время работы: лучшее – O(nk), среднее – O(nk), худшее – O(nk);
- затраты памяти – O(n + k);
- устойчивая;
- количество обменов – обычно O(nk) обменов;

Плюсы

- линейная сложность алгоритма;
- независимо от входных данных стабильное время работы;
- наиболее быстрая сортировка для массива, отсортированного в обратную сторону;
- хороша для сортировки строк;

Минусы

- нужно O(n + k) дополнительной памяти;
- по умолчанию не может обрабатывать отрицательные числа;
- по умолчанию не может обрабатывать числа с плавающей точкой;

Java Platform

1. JRE - назначение, состав

Java Runtime Enviroment (JRE) – минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений, без компилятора и других средств разработки. Состоит из Java Virtual Machine и библиотеки Java-классов.

Java Virtual Machine (JVM) – виртуальная машина Java, является основной частью **JRE**. **JVM** интерпретирует байт-код Java, предварительно созданный из исходного текста Java-программы компилятором (javac). **JVM** может также использоваться для выполнения программ, написанных на других языках программирования (Kotlin, Scala и т.п.).

2. JDK - назначение, состав

Java Development Kit – по сути является комплектом разработчика приложений на языке Java, включает в себя компилятор Java (javac), стандартные библиотеки классов Java, примеры, документацию, утилиты и **JRE**.

Java Runtime Enviroment (JRE) – минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений, без компилятора и других средств разработки. Состоит из Java Virtual Machine и библиотеки Java-классов.

Java Virtual Machine (JVM) – виртуальная машина Java, является основной частью **JRE**. **JVM** интерпретирует байт-код Java, предварительно созданный из исходного текста Java-программы компилятором (javac). **JVM** может также использоваться для выполнения программ, написанных на других языках программирования (Kotlin, Scala и т.п.).

3. 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) (5)

- 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) (4)

- 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.

4. Компиляция и запуск в консольном режиме

Сначала нужно скомпилировать класс, то есть перевести его в байт-код. Для этого в состав JDK входит компилятор Java кода javac:

E:\EJC>javac -sourcepath src -d out src/tasks/task_01/Main.java

-sourcepath задает каталог исходного кода, что позволяет компилятору найти не только тот класс, который мы отправили на компиляцию, но и найти в каталоге исходного кода другие классы, в т.ч. классы из других пакетов, которые используются в классе, отправленном на компиляцию.

С помощью **-d** мы отметили директорию, в которой будут лежать скомпилированные классы. Именно в **out** компилятор автоматически создает структуру каталогов идентичную структуре каталогов в **src**. Далее компилятор находит класс, который мы отправили на компиляцию **Main.java**, и компилируется он и все классы, которые он использует, в байткод в соответствующие каталоги в каталоге **out**.

Теперь мы можем запустить нашу программу с помощью java:

E:\EJC>java -cp out tasks.task_01.Main

С помощью **-cp** (или **-classpath**) указываем из какого каталога мы загружаем файлы, указываем класс, который запускаем.

Чтобы не прописывать полный путь к **java** и **javac** в консоли, можно задать переменную среды окружения. Если этого не делать, то придется полностью прописывать путь размещения **javac** и **java**.

5. Java Heap: принципы работы, структура

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**. Принцип работы памяти **heap** я описываю на примере применения одного из сборщиков мусора HotSpot VM – Serial GC. Есть и другие сборщики мусора, принцип работы которых отличается в той или иной мере.

Heap (Куча) (4)

- 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.

Heap делится на 2 части:

- **Young Generation Space тоже делится на 2 части:**
  - **Eden Space** – область памяти, в которую попадают только созданные объекты, после сборки мусора эта область памяти должна освободиться полностью, а выжившие объекты перемещаются в Survivor Space
  - **Survivor Space** – область памяти, которая обычно подразделяется на две подчасти («from-space» и «to-space»), между которыми объекты перемещаются по следующему принципу:
    - «from-space» постепенно заполняется объектами из Eden после сборки мусора;
    - возникает необходимость собрать мусор в «from-space»;
    - работа приложения приостанавливается и запускается сборщик мусора;
    - все живые объекты «from-space» копируются в «to-space»;
    - после этого «from-space» полностью очищается;
    - «from-space» и «to-space» меняются местами («to-space» становится «from-space» и наоборот);
    - все объекты, пережившие определенное количество перемещений между двумя частями Survivor Space перемещаются в Old Generation;
- **Old Generation Space** – область памяти, в которую попадают долгоживущие объекты, пережившие определенное количество сборок мусора. Работает по принципу перемещения живых объектов к началу «old generation space», таким образом мусор остается в конце (мусор не очищается, поверх него записываются новые объекты),

имеется указатель на последний живой объект, для дальнейшей аллокации памяти указатель просто сдвигается к концу «old generation space»;

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.

6. Представление о JIT

JIT (Just-In-Time) компиляция – динамическая компиляция, технология увеличения производительности посредством компиляции байт-кода в машинный код во время выполнения программы с применением различных оптимизаций. Благодаря этому достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом. **Основная цель JIT** – приблизиться в производительности к языкам со статической компиляцией (C/C++). С появлением **JIT** Java стала несколько более производительной.

Пример оптимизации **JIT**:

Некий метод исполняется очень часто, поэтому этот метод передается в **JIT**, он компилируется, все последующие вызовы этого метода будет кэшироваться и вызываться без перекомпиляции.

JIT позволяет достигать более высокой скорости выполнения, сохраняя одно из главных преимуществ Java-языка – переносимость. Несмотря на то, что языки со статической компиляцией все равно выигрывают в скорости, статическая компиляция лишает эти языки той переносимости, которая есть в Java.

7. Entry point в Java классе: назначение, структура (точка входа, метод main)

Каждому приложению требуется точка входа, чтобы Java знала, откуда начать выполнение кода. В Java такой точкой входа является метод main(), который имеет следующую сигнатуру:

public static void main(String[] args)
public static void main(String... args)

По крайней мере один класс в программе должен иметь метод main(). В программе может быть несколько точек входа, но начинается выполнение кода всегда с какой-то одной. Аргументы командной строки передаются в метод **main()** в массив строк **args** (называться он может иначе, это не принципиально), эти аргументы могут быть каким-то образом использованы в методе **main()**.

В методе **main()** обычно описывают самый основной алгоритм работы, например, создание каких-то объектов и вызов их методов, но конкретная программная реализация подсчетов и прочего обычно находится вне этого метода, скорее этот метод лучше использовать для консолидации функциональности других классов, которые созданы для решения определенных задач.

8. Распространение Java программ: состав, инструменты создания и способы использования jar

Для хранения классов языка Java, связанных с ними ресурсов и конфигурационных файлов в языке Java используются сжатые архивные **jar-файлы**.

Полноценно работать с технологией **JAR** можно как с помощью утилиты **jar**, входящей в состав JDK, так и с использованием классов **JAR API**. Для работы с архивами в спецификации Java есть два пакета – **java.util.zip** и **java.util.jar** соответственно для архивов **zip** и **jar**. Различие форматов заключается только в расширении архива zip.

Java-приложения гораздо удобнее собирать в один **jar-файл**, нежели хранить сложную структуру из нескольких тысяч файлов.

Такой подход позволяет **(4)**:

- подписывать содержимое jar-файла, повышая таким образом уровень безопасности;

- сокращать объем приложения;
- упростить процесс создания библиотеки;
- осуществлять контроль версий.

Так же существуют системы и утилиты для автоматизации сборки программных продуктов:

- **Apache Ant**
- **Apache Maven**
- **Gradle**