Вопросы и ответы на собеседование по основам Java

(третья часть)

61). Объясните важность блока finally в Java?

1. **Очистка ресурсов**: Один из наиболее распространенных сценариев использования блока finally - это освобождение ресурсов, таких как закрытие файлов, баз данных или сетевых соединений. Это важно для предотвращения утечек ресурсов и эффективного использования системных ресурсов.
2. **Гарантированное выполнение**: Блок finally гарантирует выполнение кода в нем, даже если произошло исключение. Это полезно, когда вам нужно выполнить некоторые завершающие операции, независимо от исключений, возникших внутри блока try.
3. **Обработка исключений**: Если исключение возникло в блоке try и не было обработано в блоке catch, блок finally может предоставить место для логирования исключения или выполнения других действий по обработке ошибки.
4. **Конечные операции**: В некоторых случаях вам может потребоваться выполнить определенные операции перед завершением выполнения программы или метода. Блок finally - это место, где вы можете разместить такие операции.

62). Можем ли мы иметь какой–либо код между блоками try и catch?

Нет, между блоком try и блоком catch не следует размещать код. Блок catch должен непосредственно следовать за блоком try. Попытка вставить код между ними вызовет ошибку компиляции, как показано в примере:

```
try{
  // код
}
System.out.println("Одна строка кода"); // недопустимо
catch(Exception e){
  //
}
```

Вместо этого код, который вы хотите выполнить после блока try, должен быть размещен за пределами этого блока, либо в блоке catch, либо в блоке finally, в зависимости от вашего намерения .

63). Можем ли мы иметь какой–нибудь код между блоками try и finally?

Нет, между блоком try и блоком finally также не следует размещать код. Блок finally должен непосредственно следовать за блоком try или, если есть, за блоком catch. Попытка вставить код между ними вызовет ошибку компиляции, как показано в примере:

```
try{
  // код
}
System.out.println("Одна строка кода"); // недопустимо
finally{
  //
}
```

Также важно отметить, что блок finally всегда будет выполняться, независимо от того, возникло исключение или нет, и он предназначен для выполнения кода, который должен быть завершен независимо от исключительных ситуаций.

64). Можем ли мы перехватить более одного исключения в одном блоке catch?

Да, начиная с Java 7, мы можем перехватывать более одного исключения с помощью одного блока catch. Этот способ обработки исключений уменьшает дублирование кода.
Важно отметить, что при перехвате более одного исключения в одном блоке catch, параметр в catch неявно объявляется как **final**. Это означает, что мы не можем присвоить значение или изменить параметр catch внутри блока catch. Вот пример:

```
catch(ArrayIndexOutOfBoundsException | ArithmeticException e)
{
   // e является final и не может быть изменен
}
```

Это позволяет упростить обработку нескольких исключений и сделать код более компактным.

65). Что такое проверяемые исключения?

Проверяемые исключения (**checked exceptions**) в Java - это подклассы класса **Throwable**, за исключением классов Error, RuntimeException и их подклассов. Проверяемые исключения должны быть либо объявлены с помощью ключевого слова throws, либо обработаны в блоке try-catch, иначе программа не будет компилироваться и возникнет ошибка компиляции.

Примеры проверяемых исключений включают:

1. IOException
2. SQLException
3. FileNotFoundException
4. InvocationTargetException
5. CloneNotSupportedException
6. ClassNotFoundException
7. InstantiationException

Для этих исключений требуется явно обрабатывать ошибки или указывать, что метод может бросать это исключение, с помощью ключевого слова **throws**. Это помогает обеспечить более надежное управление ошибками в программе.

66). Что такое непроверяемые исключения?

Непроверяемые исключения (**unchecked exceptions**) в Java - это все подклассы класса **RuntimeException**. Они называются непроверяемыми, потому что компилятор не проверяет, обрабатывает ли метод исключение или нет. Программа будет компилироваться даже в том случае, если исключение не обрабатывается явно с использованием блока try-catch или ключевого слова throws.

Если исключение возникает в программе и не обрабатывается, программа завершается. Эти исключения могут быть сложными для обработки, так как причины их возникновения могут быть в разных частях программы.

Примеры непроверяемых исключений включают:

1. ArithmeticException
2. ArrayIndexOutOfBoundsException
3. ClassCastException
4. IndexOutOfBoundsException
5. NullPointerException
6. NumberFormatException
7. StringIndexOutOfBoundsException
8. UnsupportedOperationException

Обработка непроверяемых исключений необязательна, но она может быть важной для обеспечения надежности программы и предотвращения непредсказуемых сбоев.

67) В чем разница между проверяемыми (checked) и непроверяемыми (unchecked) исключениями в Java?

UNCHECKED EXCEPTION	CHECKED EXCEPTION
1) Все подклассы класса RuntimeException в Java называются непроверяемыми исключениями.	1) Все подклассы класса Throwable, кроме RuntimeException, называются проверяемыми исключениями.
2) Непроверяемые исключения не нужно обрабатывать на этапе компиляции.	2) Проверяемые исключения должны быть обработаны на этапе компиляции.
3) Эти исключения в основном возникают из-за ошибок в логике программы или неправильного использования методов и объектов, таких как деление на ноль или попытка обращения к объекту, который не был инициализирован (null).	3) Эти исключения в основном возникают из-за внешних факторов и операций, которые могут привести к проблемам, которые программист должен предвидеть и обработать.
4) ArrayIndexOutOfBoundsException, ClassCastException, IndexOutOfBoundException	4) SqlException, FileNotFoundException,ClassNotFoundException

68). Что такое обработка исключений по умолчанию в Java?

В Java, когда JVM обнаруживает код, который может вызвать исключение, она создает новый объект обработки исключений, включая следующую информацию:

1. Название исключения.
2. Описание исключения.
3. Местоположение исключения.

После создания объекта JVM проверяет, есть ли в коде обработка исключений. Если есть обработка исключений, то она обрабатывает исключение и продолжает выполнение программы. Если обработки исключений нет, JVM передает обработку исключения обработчику по умолчанию и завершает программу аварийно.

Обработчик по умолчанию выводит описание исключения, печатает стек вызовов (**stacktrace**) и местоположение исключения, а затем завершает программу.

Важно отметить, что основным недостатком этого механизма обработки исключений по умолчанию является **аварийное завершение программы.**

69). Объясните ключевое слово throw в Java?

В Java, обычно исключения выбрасываются JVM, и мы обрабатываем исключения с помощью блока **try-catch**. Однако есть ситуации, когда нам нужно выбрасывать пользовательские исключения или исключения времени выполнения. В таком случае мы используем ключевое слово **throw** для явного выбрасывания исключения.
Синтаксис: **throw throwableInstance;**
throwableInstance должен быть типа Throwable или любого из его подклассов. После выполнения оператора throw выполнение останавливается, и последующие операторы не выполняются.
Как только throwableInstance выбрасывается, JVM проверяет, есть ли блок catch для обработки исключения. Если блока catch нет, JVM переходит к следующему блоку catch до тех пор, пока не найдет соответствующий обработчик. Если соответствующий обработчик не найден, то обработчик исключений по умолчанию останавливает программу и выводит описание и местоположение исключения.

В итоге мы используем ключевое слово **throw** для явного выбрасывания исключения.

70). Можем ли мы написать какой-нибудь код после оператора throw?

После оператора throw выполнение JVM останавливается, и последующие операторы не выполняются.
Если мы попытаемся написать какой-либо оператор после оператора throw, мы получим ошибку времени компиляции, говорящую о недостижимом коде (**unreachable code)**.

71). Объясните важность ключевого слова throws в Java?

Ключевое слово **throws** применяется только к проверяемым исключениям (checked exceptions), а непроверяемые исключения (unchecked exceptions) не требуют его использования.
throws используется в конце сигнатуры метода, чтобы указать, что из метода может быть сгенерировано исключение заданного типа.
Важность ключевого слова throws заключается в следующем:

1. **Делегирование обработки исключений**: Когда метод генерирует проверяемое исключение (checked exception), он может использовать ключевое слово throws для указания, что он не будет обрабатывать это исключение внутри себя. Вместо этого, метод говорит вызывающему коду, что это исключение должно быть обработано вызывающим методом. Это позволяет разделять ответственность за обработку исключений между разными частями программы.
2. **Документирование исключений**: Ключевое слово throws также служит документацией для других разработчиков. Оно указывает, какие исключения может сгенерировать метод, что помогает другим разработчикам понимать, какие ошибки могут возникнуть при вызове метода, и как нужно обрабатывать эти ошибки.
3. **Предотвращение неожиданных ошибок**: Если метод не использует ключевое слово throws и генерирует проверяемое исключение, это может привести к неожиданной ошибке в коде, если вызывающий метод не предпринимает соответствующие меры для обработки исключения. Использование ключевого слова throws помогает предотвратить такие ситуации и обеспечивает более предсказуемое поведение программы.
4. **Соблюдение соглашений**: Использование ключевого слова throws является частью соглашений о структуре кода в Java. Это делает код более читаемым и понятным для других разработчиков, которые могут работать с этим кодом.

72). Объясните важность оператора finally по отношению к оператору return

Оператор finally в Java выполняется после завершения блока try или catch, независимо от того, было ли исключение или нет. Это означает, что оператор finally предоставляет возможность выполнить определенные действия независимо от исключительных ситуаций. Это имеет важное значение по отношению к оператору return по следующим причинам:

1. **Гарантированное выполнение**: finally гарантирует выполнение кода, даже если произошло исключение. Это полезно, например, для освобождения ресурсов, таких как закрытие файлов, потоков или соединений с базой данных, независимо от того, произошло исключение или нет.
2. **Влияние на возвращаемое значение:** Если внутри блока finally есть оператор return, он может заменить возвращаемое значение метода, даже если в блоке try было другое возвращаемое значение. Это происходит потому, что блок finally выполняется после завершения блока try и может модифицировать возвращаемое значение.

Пример:

```
public int someMethod() {
    try {
        // Какой-то код
        return 1;
    } finally {
        // Блок finally выполняется всегда
        // и может изменить возвращаемое значение
        return 2;
    }
}
```

В этом примере метод someMethod вернет значение 2 из-за блока finally, несмотря на то, что внутри блока try было возвращено значение 1.

73). Объяснение ситуации, когда блок finally не будет выполнен

Блок finally не будет выполнен, когда JVM завершает свою работу. Если мы используем **System.exit(0)** в блоке try, то блок finally, если он присутствует, не будет выполнен.

74). Можем ли мы использовать оператор catch для проверяемых исключений?

Если нет возможности вызвать исключение в нашем коде, то мы не можем объявлять блок catch для обработки проверяемых исключений. Это вызывает ошибку времени компиляции, если мы попытаемся обработать проверяемые исключения, когда нет возможности вызвать исключение.

75). Как создавать пользовательские (user defined) исключения в Java?

Для создания пользовательских сообщений об ошибках мы используем пользовательские исключения. Мы можем создавать пользовательские исключения как проверяемые, так и непроверяемые.
Мы можем создать пользовательские исключения, которые расширяют класс **Exception** или его подклассы, чтобы пользовательское исключение стало проверяемым.
Пользовательские исключения могут расширять класс **RuntimeException** для создания пользовательских непроверяемых исключений.

Примечание: рекомендуется держать наши пользовательские классы исключений как непроверяемые, то есть мы должны расширить класс RuntimeException, а не Excpetion class.

76). Можем ли мы повторно вызвать то же исключение из обработчика catch?

Да, мы можем повторно вызвать. Если мы хотим повторно вызвать проверяемое исключение из блока catch, нам нужно объявить это исключение.

77). Можем ли мы использовать вложенные операторы try в Java?

Да, операторы try могут быть вложенными. Мы можем объявлять операторы try внутри блока другого оператора try.

78). Объясните важность класса Throwable и его методов?

Класс Throwable является корневым классом для исключений. Все исключения производны от этого класса Throwable .

Два основных подкласса Throwable - это Exception и Error.

Три метода, определенные в классе Throwable:

- void **printStackTrace()**: Этот метод выводит информацию об исключении в следующем формате: имя исключения, описание, а затем стек вызовов.
- **getMessage()** Этот метод выводит только описание исключения.
- **toString()**: Он выводит имя и описание исключения.

79). Объясните, когда будет ClassNotFoundException?

Исключение ClassNotFoundException будет сгенерировано, когда JVM попытается загрузить класс по его имени и не сможет найти этот класс. Примером такого исключения является опечатка в имени класса, когда мы пытаемся загрузить класс по его строковому имени, и класс не может быть найден, что вызывает ClassNotFoundException.

80). Объясните, когда будет NoClassDefFoundError?

Эта ошибка возникает, когда JVM пытается загрузить класс, но не находит определения этого класса - тогда возникает NoClassDefFoundError. Класс может существовать на этапе компиляции, но не может быть найден на этапе выполнения. Это может быть вызвано опечаткой в имени класса при вводе командной строки, неправильным указанием пути поиска классов (classpath), или отсутствием файла с байт-кодом класса.

81). Что такое процесс?

Процесс - это выполняющийся экземпляр программы в памяти компьютера. Каждый процесс имеет свое собственное адресное пространство и системные ресурсы, и он функционирует независимо от других процессов.
Например, при запуске программы на Java, процесс — это её среда исполнения (JRE).
Процесс может состоять из одного или нескольких потоков, где каждый поток выполняет определенный набор инструкций в рамках процесса.
Процессы считаются "тяжеловесными", поскольку они требуют собственной памяти и ресурсов.

Процессы - это фундаментальное понятие в операционных системах и обеспечивают изоляцию между разными приложениями и задачами, выполняющимися на компьютере.

82). Что такое поток(thread) в Java?

Поток (thread) в Java представляет собой отдельный путь выполнения в программе.

Основные характеристики потоков в Java включают:

1. **Легковесность**: Потоки являются легковесными единицами выполнения, что означает, что они требуют меньше ресурсов и создаются и завершаются более эффективно, чем процессы.
2. **Совместное использование адресного пространства**: Все потоки, принадлежащие одному процессу, разделяют общее адресное пространство процесса. Это позволяет им обмениваться данными и ресурсами.
3. **Простота создания**: Создание потоков в Java относительно просто и может быть выполнено с помощью стандартных библиотечных классов.
4. **Параллельное выполнение**: Потоки могут выполняться параллельно, что позволяет программам эффективно использовать многоядерные процессоры и улучшать производительность.
5. **Отзывчивость:** Использование потоков позволяет создавать отзывчивые приложения, которые могут обрабатывать множество задач одновременно.

Потоки в Java можно создавать и управлять с помощью классов из пакета **java.lang.Thread** или с использованием средств более высокого уровня, таких как **java.util.concurrent**.

83) В чем разница между процессом и потоком?

PROCESS	THREAD
1) Исполняемая программа.	1) Отдельный путь выполнения в программе.
2) Процессы считаются "тяжеловесными".	2) Потоки являются "легковесными".
3) Процессы требуют отдельного адресного пространства.	3) Потоки используют одно и то же адресное пространство..
4) Межпроцессное взаимодействие является дорогостоящим.	4) Связь между потоками обходится дешевле по сравнению с процессами.
5) Переключение контекста с одного процесса на другой является дорогостоящим.	5) Переключение контекста между потоками обходится недорого.

84). Что такое многозадачность (multitasking)?

Multitasking в контексте компьютеров означает выполнение более чем одной задачи или действия одновременно на компьютере. Например, это может включать в себя одновременное использование электронных таблиц и калькулятора.
Когда компьютер способен выполнять несколько задач одновременно, это увеличивает эффективность использования ресурсов и позволяет пользователям выполнять разные операции параллельно, улучшая их производительность.

85). Какие существуют типы многозадачности?

Существует два различных типа многозадачности:

1. Многозадачность на основе процессов
2. Многозадачность на основе потоков

Многозадачность на основе процессов: позволяет запускать две или более программы параллельно. В многозадачности на основе процессов процесс является самой маленькой частью кода.

Пример: одновременный запуск Microsoft Word и Microsoft PowerPoint.

Многозадачность на основе потоков: позволяет запускать части программы параллельно.

Пример: форматирование текста и печать документа в Word одновременно.

Java поддерживает многозадачность на основе потоков и предоставляет встроенную поддержку многопоточности.

86). Каковы преимущества многопоточного программирования?

1. **Максимальное использование процессорного времени**: Многопоточное программирование позволяет использовать неиспользуемое процессорное время для выполнения других задач, что приводит к более быстрому выполнению программы. В однопоточной среде каждая задача должна быть завершена перед переходом к следующей, что приводит к простою процессора.
2. **Улучшение отзывчивости**: В наличии нескольких потоков приложение может продолжать работу, даже если один из потоков занят выполнением долгой операции. Это делает приложение более отзывчивым к действиям пользователя.
3. **Эффективное использование ресурсов**: Многопоточные программы более эффективно используют доступные ресурсы, такие как процессор и память, по сравнению с однопоточными программами.
4. **Параллельное выполнение задач:** Многопоточное программирование позволяет параллельно выполнять задачи, что особенно полезно для операций, требующих одновременной обработки, таких как сетевые запросы и обработка данных.
5. **Улучшение производительности**: В многопоточных приложениях можно достичь значительного повышения производительности, особенно на многоядерных процессорах, где задачи могут выполняться параллельно.

Однако многопоточное программирование также может быть сложным с точки зрения управления потоками и синхронизации доступа к ресурсам, и требует внимательного проектирования и разработки.

87). Перечислите Java API, поддерживающий потоки.

1. **java.lang.Thread**: Это один из способов создания потока. Путем расширения класса Thread и переопределения метода run() мы можем создать поток в Java.
2. **java.lang.Runnable**: Runnable - это интерфейс в Java. Путем реализации интерфейса Runnable и переопределения метода run() мы можем создать поток в Java.
3. **java.lang.Object**: Класс Object является суперклассом для всех классов в Java. В классе Object у нас есть три метода **wait()**, **notify()** и **notifyAll()**, которые поддерживают потоки.
4. **java.util.concurrent**: Этот пакет содержит классы и интерфейсы, поддерживающие параллельное программирование. Например, интерфейс **Executor**, класс **FutureTask** и многие другие.

88). Объясните про основной поток (main thread) в Java.

Основной поток - это первый поток, который запускается непосредственно после запуска программы. Основной поток важен, потому что:

- Все дочерние потоки порождаются из основного потока.
- Метод main является последним, который завершает выполнение. Когда JVM вызывает метод main(), он запускает новый поток. Метод main() временно приостанавливается, пока начинает выполняться новый поток(new thread).

89). Какими способами можно создавать потоки в Java?

В Java потоки можно создавать двумя способами:

1) Путем расширения класса Thread

```
public class MyClass extends Thread {
@Override
public void run() {
//
}
}
```

2) Путем реализации интерфейса Runnable.

```
public class MyClass implements Runnable {
@Override
public void run() {
//
}
}
```

90). Какой подход к созданию потока лучше всего?

Самый лучший способ создания потоков в Java - это реализация интерфейса **Runnable**.

Вот несколько причин, почему реализация Runnable предпочтительнее:

1. **Гибкость**: При реализации интерфейса Runnable ваш класс может по-прежнему расширять другие классы или реализовывать другие интерфейсы, что дает больше гибкости в иерархии классов.
2. **Общий доступ к ресурсам**: Реализация Runnable способствует лучшему разделению обязанностей. Вы можете создать несколько потоков, которые используют один и тот же экземпляр Runnable, что позволяет эффективно делить ресурсы между потоками.
3. **Инкапсуляция**: Это способствует лучшей инкапсуляции поведения вашего потока. Метод run() представляет собой задачу, которую должен выполнить поток, делая ваш код более модульным и легким в обслуживании.
4. **Соответствие**: Этот подход соответствует иерархии классов Java, так как вы не вводите дополнительного уровня наследования с классом Thread.

Хотя расширение класса Thread также является допустимым способом создания потоков, обычно рекомендуется использовать подход с Runnable по перечисленным выше причинам.