Исправляем 7 распространенных ошибок обработки исключений в Java

Привет, Хабр! Представляю вашему вниманию перевод статьи [Fixing 7 Common Java Exception Handling Mistakes](https://dzone.com/articles/fixing-7-common-java-exception-handling-mistakes) автора Thorben Janssen.  
  
Обработка исключения является одной из наиболее распространенных, но не обязательно одной из самых простых задач. Это все еще одна из часто обсуждаемых тем в опытных командах, и есть несколько передовых методов и распространенных ошибок, о которых вы должны знать.  
  
Вот несколько вещей, которые следует избегать при обработке исключений в вашем приложении.

Ошибка 1: объявление java.lang.Exception или java.lang.Throwable

Ошибка 2: перехват обобщенных исключений

Ошибка 3: Логирование и проброс исключений

Ошибка 4: использование исключений для управления потоком

Ошибка 5: удалить причину возникновения исключения

Ошибка 6: Обобщение исключений

Ошибка 7: добавление ненужных преобразований исключений

**Ошибка 1: объявление java.lang.Exception или java.lang.Throwable**

Как вы уже знаете, вам нужно либо объявить, либо обработать проверяемое исключение. Но проверяемые исключения — это не единственные, которые вы можете указать. Вы можете использовать любой подкласс java.lang.Throwable в предложении throws. Таким образом, вместо указания двух разных исключений, которые выбрасывает следующий фрагмент кода, вы можете просто использовать исключение java.lang.Exception в предложении throws.

**public** **void** **doNotSpecifyException**() **throws** Exception {

doSomething();

}

**public** **void** **doSomething**() **throws** NumberFormatException, IllegalArgumentException {

// do something

}

Но это не значит, что вы должны это сделать. Указание Exeption или Throwable делает почти невозможным правильное обращение с ними при вызове вашего метода.Единственная информация, которую получает вызывающий вами метод, заключается в том, что что-то может пойти не так. Но вы не делитесь какой-либо информацией о каких-либо исключительных событиях, которые могут произойти. Вы скрываете эту информацию за обобщенными причинами выброса исключений.Становится еще хуже, когда ваше приложение меняется со временем. Выброс обобщенных исключений скрывает все изменения исключений, которые вызывающий должен ожидать и обрабатывать. Это может привести к нескольким непредвиденным ошибкам, которые необходимо найти в тестовом примере вместо ошибки компилятора.

Используйте конкретные классы

Гораздо лучше указать наиболее конкретные классы исключений, даже если вам приходится использовать несколько из них. Это сообщает вызывающему устройству, какие исключительные событий нужно обрабатывать. Это также позволяет вам обновить предложение throw, когда ваш метод выдает дополнительное исключение. Таким образом, ваши клиенты знают об изменениях и даже получают ошибку, если вы изменяете выбрасываемые исключения. Такое исключение намного проще найти и обработать, чем исключение, которое появляется только при запуске конкретного тестового примера.

**public** **void** **specifySpecificExceptions**() **throws** NumberFormatException, IllegalArgumentException {

doSomething();

}

**Ошибка 2: перехват обобщенных исключений**

Серьезность этой ошибки зависит от того, какой программный компонент вы реализуете, и где вы обнаруживаете исключение. Возможно, было бы хорошо поймать java.lang.Exception в основном методе вашего приложения Java SE. Но вы должны предпочесть поймать определенные исключения, если вы реализуете библиотеку или работаете над более глубокими слоями вашего приложения.  
  
Это дает несколько преимуществ. Такой подход позволяет обрабатывать каждый класс исключений по-разному и не позволяет вам перехватывать исключения, которых вы не ожидали.  
  
Но имейте в виду, что первый блок catch, который обрабатывает класс исключения или один из его супер-классов, поймает его. Поэтому сначала обязательно поймайте наиболее специфический класс. В противном случае ваши IDE покажут сообщение об ошибке или предупреждении о недостижимом блоке кода.

**try** {

doSomething();

} **catch** (NumberFormatException e) {

// handle the NumberFormatException

log.error(e);

} **catch** (IllegalArgumentException e) {

// handle the IllegalArgumentException

log.error(e);

}

**Ошибка 3: Логирование и проброс исключений**

Это одна из самых популярных ошибок при обработке исключений Java. Может показаться логичным регистрировать исключение там, где оно было брошено, а затем пробросить его вызывающему объекту, который может реализовать конкретную обработку для конкретного случая использования. Но вы не должны делать это по трем причинам:  
  
1. У вас недостаточно информации о прецеденте, который хочет реализовать вызывающий объект вашего метода. Исключение может быть частью ожидаемого поведения и обрабатываться клиентом. В этом случае нет необходимости регистрировать его. Это добавит ложное сообщение об ошибке в файл журнала, который должен быть отфильтрован вашей операционной группой.  
  
2. Сообщение журнала не предоставляет никакой информации, которая еще не является частью самого исключения. Его трассировка и трассировка стека должны содержать всю необходимую информацию об исключительном событии. Сообщение описывает это, а трассировка стека содержит подробную информацию о классе, методе и строке, в которой она произошла.  
  
3. Вы можете регистрировать одно и то же исключение несколько раз, когда вы регистрируете его в каждом блоке catch, который его ловит. Это испортит статистику в вашем инструменте мониторинга и затрудняет чтение файла журнала для ваших операций и команды разработчиков.

Регистрируйте исключение там, где вы его обрабатываете

Таким образом, лучше всего регистрировать исключение тогда, когда вы его обрабатываете. Как в следующем фрагменте кода. Метод doSomething генерирует исключение. Метод doMore просто указывает его, потому что у разработчика недостаточно информации для его обработки. Затем он обрабатывается в методе doEvenMore, который также записывает сообщение журнала.

**public** **void** **doEvenMore**() {

**try** {

doMore();

} **catch** (NumberFormatException e) {

// handle the NumberFormatException

} **catch** (IllegalArgumentException e) {

// handle the IllegalArgumentException

}

}

**public** **void** **doMore**() **throws** NumberFormatException, IllegalArgumentException {

doSomething();

}

**public** **void** **doSomething**() **throws** NumberFormatException, IllegalArgumentException {

// do something

}

**Ошибка 4: использование исключений для управления потоком**

Использование исключений для управления потоком вашего приложения считается анти-шаблоном по двум основным причинам:  
  
Они в основном работают как оператор Go To, потому что они отменяют выполнение блока кода и переходят к первому блоку catch, который обрабатывает исключение. Это делает код очень трудным для чтения.  
  
Они не так эффективны, как общие структуры управления Java. Как видно из названия, вы должны использовать их только для исключительных событий, а JVM не оптимизирует их так же, как и другой код.Таким образом, лучше использовать правильные условия, чтобы разбить свои циклы или инструкции if-else, чтобы решить, какие блоки кода должны быть выполнены.

**Ошибка 5: удалить причину возникновения исключения**

Иногда вам может понадобиться обернуть одно исключение в другое. Возможно, ваша команда решила использовать специальное исключение для бизнеса с кодами ошибок и единой обработкой. Нет ничего плохого в этом подходе, если вы не устраните причину.  
  
Когда вы создаете новое исключение, вы всегда должны устанавливать первоначальное исключение в качестве причины. В противном случае вы потеряете трассировку сообщения и стека, которые описывают исключительное событие, вызвавшее ваше исключение. Класс Exception и все его подклассы предоставляют несколько методов-конструкторов, которые принимают исходное исключение в качестве параметра и задают его как причину.

**try** {

doSomething();

} **catch** (NumberFormatException e) {

**throw** **new** MyBusinessException(e, ErrorCode.CONFIGURATION\_ERROR);

} **catch** (IllegalArgumentException e) {

**throw** **new** MyBusinessException(e, ErrorCode.UNEXPECTED);

}

**Ошибка 6: Обобщение исключений**

Когда вы обобщаете исключение, вы ловите конкретный, например, NumberFormatException, и вместо этого генерируете неспецифическое java.lang.Exception. Это похоже, но даже хуже, чем первая ошибка, которую я описал в этой статье. Он не только скрывает информацию о конкретном случае ошибки на вашем API, но также затрудняет доступ.

**public** **void** **doNotGeneralizeException**() **throws** Exception {

**try** {

doSomething();

} **catch** (NumberFormatException e) {

**throw** **new** Exception(e);

} **catch** (IllegalArgumentException e) {

**throw** **new** Exception(e);

}

}

Как вы можете видеть в следующем фрагменте кода, даже если вы знаете, какие исключения может вызвать метод, вы не можете просто их поймать. Вам нужно поймать общий класс Exception и затем проверить тип его причины. Этот код не только громоздкий для реализации, но его также трудно читать. Становится еще хуже, если вы сочетаете этот подход с ошибкой 5. Это удаляет всю информацию об исключительном событии.

**try** {

doNotGeneralizeException();

} **catch** (Exception e) {

**if** (e.getCause() **instanceof** NumberFormatException) {

log.error("NumberFormatException: " + e);

} **else** **if** (e.getCause() **instanceof** IllegalArgumentException) {

log.error("IllegalArgumentException: " + e);

} **else** {

log.error("Unexpected exception: " + e);

}

}

Итак, какой подход лучший?

Будьте конкретны и сохраняйте причину возникновения исключения.

Исключения, которые вы бросаете, должны всегда быть максимально конкретными. И если вы оборачиваете исключение, вы также должны установить исходный исключение в качестве причины, чтобы не потерять трассировку стека и другую информацию, описывающую исключительное событие.

**try** {

doSomething();

} **catch** (NumberFormatException e) {

**throw** **new** MyBusinessException(e, ErrorCode.CONFIGURATION\_ERROR);

} **catch** (IllegalArgumentException e) {

**throw** **new** MyBusinessException(e, ErrorCode.UNEXPECTED);

}

**Ошибка 7: добавление ненужных преобразований исключений**

Как я уже объяснял ранее, может быть полезно обернуть исключения в пользовательские, если вы установите исходное исключение в качестве причины. Но некоторые архитекторы переусердствуют и вводят специальный класс исключений для каждого архитектурного уровня. Таким образом, они улавливают исключение в уровне персистентности и переносят его в MyPersistenceException. Бизнес-уровень ловит и обертывает его в MyBusinessException, и это продолжается до тех пор, пока оно не достигнет уровня API или не будет обработано.

**public** **void** **persistCustomer**(Customer c) **throws** MyPersistenceException {

// persist a Customer

}

**public** **void** **manageCustomer**(Customer c) **throws** MyBusinessException {

// manage a Customer

**try** {

persistCustomer(c);

} **catch** (MyPersistenceException e) {

**throw** **new** MyBusinessException(e, e.getCode());

}

}

**public** **void** **createCustomer**(Customer c) **throws** MyApiException {

// create a Customer

**try** {

manageCustomer(c);

} **catch** (MyBusinessException e) {

**throw** **new** MyApiException(e, e.getCode());

}

}

Легко видеть, что эти дополнительные классы исключений не дают никаких преимуществ. Они просто вводят дополнительные слои, которые оборачивают исключение. И хотя было бы забавно обернуть подарок во множестве красочной бумаги, это не очень хороший подход к разработке программного обеспечения.

Обязательно добавьте информацию

Просто подумайте о коде, который должен обрабатывать исключение или о самом себе, когда вам нужно найти проблему, вызвавшую исключение. Сначала вам нужно прорваться через несколько уровней исключений, чтобы найти исходную причину. И до сегодняшнего дня я никогда не видел приложение, которое использовало этот подход, и добавляло полезную информацию с каждым слоем исключения. Они либо обобщают сообщение об ошибке и код, либо предоставляют избыточную информацию.  
  
Поэтому будьте осторожны с количеством настраиваемых классов исключений, которые вы вводите. Вы всегда должны спрашивать себя, дает ли новый класс исключений дополнительную информацию или другие преимущества. В большинстве случаев для достижения этого вам не требуется более одного уровня пользовательских исключений.

**public** **void** **persistCustomer**(Customer c) {

// persist a Customer

}

**public** **void** **manageCustomer**(Customer c) **throws** MyBusinessException {

// manage a Customer

**throw** **new** MyBusinessException(e, e.getCode());

}

**public** **void** **createCustomer**(Customer c) **throws** MyBusinessException {

// create a Customer

manageCustomer(c);

}