**Билет 15**

**Обработка ошибок** — одна из тех рутинных вещей, которыми нам всем приходится заниматься при программировании. Программа может получить аномальные входные данные, на устройстве могут произойти сбои. Короче говоря, выполнение программы может пойти по неверному пути, и если это случается, мы, программисты, должны позаботиться, чтобы наш код сделал то, что ему положено сделать.

Обработка ошибок важна, но если они заслоняют собой логику программы — значит, она реализована неверно.

**Используйте исключения вместо кодов ошибок**

В далеком прошлом многие языки программирования не поддерживали механизма обработки исключений. В таких языках возможности обработки и получения информации об ошибках были ограничены. Программа либо устанавливала флаг ошибки, либо возвращала код, который проверялся вызывающей стороной.

У обоих решений имеется общий недостаток: они загромождают код на стороне вызова. Вызывающая сторона должна проверять ошибки немедленно после вызова. К сожалению, об этом легко забыть. По этой причине при обнаружении ошибки лучше инициировать исключение. Код вызова становится более понятным, а его логика не скрывается за кодом обработки ошибок.

**try-catch-finally:**

У исключений есть одна интересная особенность: они определяют область видимости в вашей программе. Размещая код в секции try команды try-catch-finally, вы утверждаете, что выполнение программы может прерваться в любой точке, а затем продолжиться в секции catch. Блоки try в каком-то отношении напоминают транзакции. Секция catch должна оставить программу в целостном состоянии, что бы и произошло в секции try. По этой причине написание кода, который может инициировать исключения, рекомендуется начинать с конструкции try-catch-finally. Это поможет вам определить, чего должен ожидать пользователь кода, что бы ни произошло в коде try.

public List retrieveSection(String sectionName) {

try { FileInputStream stream = new FileInputStream(sectionName)

}

catch (Exception e) {

throw new StorageException("retrieval error", e);

}

return new ArrayList();

}

Теперь тест проходит успешно, потому что мы перехватили исключение.

**Используйте непроверяемые исключения**

Но сейчас стало ясно, что они не являются необходимыми для создания надежных программ. В C# нет проверяемых исключений, и несмотря на все доблестные попытки, в C++ они так и не появились. Их также нет в Python и Ruby. Тем не менее на всех этих языках можно писать надежные программы.

Цена проверяемых исключений — нарушение принципа открытости/закрытости.

Представьте иерархию вызовов большой системы. Функции верхнего уровня вызывают функции нижележащего уровня, которые, в свою очередь, вызывают функции низких уровней и т. д. Теперь допустим, что одна из низкоуровневых функций изменилась таким образом, что она должна инициировать исключение. Если это исключение является проверяемым, то в сигнатуру функции должна быть добавлена секция throws. Но тогда каждая функция, вызывающая нашу измененную функцию, тоже должна быть изменена с перехватом нового исключения или присоединением соответствующей секции throws к ее сигнатуре. И так до бесконечности. В итоге мы имеем каскад изменений, пробивающихся с нижних уровней программного продукта на верхние уровни! При этом нарушается инкапсуляция, потому что все функции на пути инициирования должны располагать подробной информацией об этом низкоуровневом исключении. Учитывая, что главной целью исключений является возможность обработки ошибок «на расстоянии», такое нарушение инкапсуляции проверяемыми исключениями выглядит особенно постыдно.

Проверяемые исключения иногда могут пригодиться при написании особо важных библиотек.

**Передавайте контекст с исключениями**

Каждое исключение, инициируемое в программе, должно содержать достаточно контекстной информации для определения источника и местонахождения ошибки.

Создавайте содержательные сообщения об ошибках и передавайте их со своими исключениями. Включайте в них сведения о сбойной операции и типе сбоя. Если в приложении ведется журнал, передайте информацию, достаточную для регистрации ошибки из секции catch.

**Определяйте классы исключений в контексте потребностей вызывающей стороны**

Существует много способов классификации ошибок. Например, ошибки можно классифицировать по источнику, то есть по компоненту, в котором они произошли. Также возможна классификация по типу: сбои устройств, сетевые сбои, ошибки программирования и т. д. Однако при определении классов исключений в приложениях думать необходимо прежде всего о том, как они будут перехватываться.

Рассмотрим пример неудачной классификации исключений. Далее приводится конструкция try-catch-finally для сторонней библиотечной функции. Она учитывает все исключения, которые могут быть инициированы при вызовах:

ACMEPort port = new ACMEPort(12);

try { port.open(); }

catch (DeviceResponseException e) { r

eportPortError(e); logger.log("Device response exception", e);

} catch (ATM1212UnlockedException e) {

reportPortError(e); logger.log("Unlock exception", e);

} catch (GMXError e) {

reportPortError(e); logger.log("Device response exception");

} finally { … }

Мы должны сохранить ошибку и убедиться в том, что работа программы может быть продолжена. В этом случае, поскольку выполняемая работа остается более или менее постоянной независимо от исключения, код можно существенно упростить — для этого мы создаем «обертку» для вызываемой функции API и обеспечиваем возвращение стандартного типа исключения:

LocalPort port = new LocalPort(12);

try { port.open(); }

catch (PortDeviceFailure e) {

reportError(e); logger.log(e.getMessage(), e); }

finally { … }

Класс LocalPort представляет собой простую обертку, которая перехватывает и преобразует исключения, инициированные классом ACMEPort.

Обертки — вроде той, которую мы определили для ACMEPort, — бывают очень полезными. Более того, инкапсуляция вызовов сторонних API принадлежит к числу стандартных приемов. Создавая обертку для стороннего вызова, вы сокращаете до минимума зависимость от него в своем коде: в будущем вы можете переключиться на другую библиотеку без сколько-нибудь заметных проблем. Обертки также упрощают имитацию сторонних вызовов в ходе тестирования кода.

Последнее преимущество оберток заключается в том, что вы не ограничиваетесь архитектурными решениями разработчика API. Вы можете определить тот API, который вам удобен. В предыдущем примере мы определили для всех сбоев порта один тип исключения, и код от этого стал намного чище.

**Определите нормальный путь выполнения**

В следующем, довольно неуклюжем фрагменте суммируются командировочные расходы на питание:

try {

MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());

m\_total += expenses.getTotal();

} catch(MealExpensesNotFound e) {

m\_total += getMealPerDiem();

}

Если работник предъявил счет по затратам на питание, то сумма включается в общий итог. Если счет отсутствует, то работнику за этот день начисляется определенная сумма. Исключение загромождает логику программы. А если бы удалось обойтись без обработки особого случая? Это позволило бы заметно упростить код:

MealExpenses expenses = expenseReportDAO.getMeals(employee.getID()); m\_total += expenses.getTotal();

Можно ли упростить код до такой формы? Оказывается, можно. Мы можем изменить класс ExpenseReportDAO, чтобы он всегда возвращал объект MealExpense. При отсутствии предъявленного счета возвращается объект MealExpense, у которого в качестве затрат указана стандартная сумма, начисляемая за день:

public class PerDiemMealExpenses implements MealExpenses {

public int getTotal() { // Вернуть стандартные ежедневные затраты на питание }

}

Такое решение представляет собой реализацию паттерна ОСОБЫЙ СЛУЧАЙ. Программист создает класс или настраивает объект так, чтобы он обрабатывал особый случай за него. Это позволяет избежать обработки исключительного поведения в клиентском коде. Все необходимое поведение инкапсулируется в объекте особого случая.

**Не возвращайте null**

Возвращая null, мы фактически создаем для себя лишнюю работу, а для вызывающей стороны — лишние проблемы. Стоит пропустить всего одну проверку null, и приложение «уходит в штопор».

Легко сказать, что проблемы в приведенном коде возникли из-за пропущенной проверки null. В действительности причина в другом: этих проверок слишком много. Если у вас возникает желание вернуть null из метода, рассмотрите возможность выдачи исключения или возвращения объекта «особого случая». Если ваш код вызывает метод стороннего API, способный вернуть null, создайте для него обертку в виде метода, который инициирует исключение или возвращает объект особого случая.

Довольно часто объекты особых случаев легко решают проблему. Допустим, у вас имеется код следующего вида:

List employees = getEmployees();

if (employees != null) {

for(Employee e : employees) {

totalPay += e.getPay(); }

}

Сейчас метод getEmployees может возвращать null, но так ли это необходимо? Если изменить getEmployee так, чтобы метод возвращал пустой список, код станет чище:

List employees = getEmployees();

for(Employee e : employees) {

totalPay += e.getPay();

}

**Не передавайте null**

Возвращать null из методов плохо, но передавать null при вызове еще хуже. По возможности избегайте передачи null в своем коде (исключение составляют разве что методы сторонних API, при вызове которых без нее не обойтись). Следующий пример поясняет, почему не следует передавать null. Возьмем простой метод для вычисления метрики по двум точкам:

public class MetricsCalculator {

public double xProjection(Point p1, Point p2) {

return (p2.x — p1.x) \* 1.5; }

… }

Что произойдет, если при вызове будет передан аргумент null? calculator.xProjection(null, new Point(12, 13)); Конечно, возникнет исключение NullPointerException. Как исправить его? Можно создать новый тип исключения и инициировать его в методе:

public class MetricsCalculator {

public double xProjection(Point p1, Point p2) {

if (p1 == null || p2 == null) {

throw InvalidArgumentException( "Invalid argument for MetricsCalculator.xProjection");

} return (p2.x — p1.x) \* 1.5;

}

}

В большинстве языков программирования не существует хорошего способа справиться со случайной передачей null с вызывающей стороны. А раз так, разумно запретить передачу null по умолчанию. В этом случае вы будете знать, что присутствие null в списке аргументов свидетельствует о возникшей проблеме; это будет способствовать уменьшению количества ошибок, сделанных по неосторожности.