Белорусский Государственный Университет Информатики и Радиоэлектроники
кафедра интеллектуальных информационных технологий

ЭКЗАМЕН

по курсу «Проектирование программ в интеллектуальных системах»
(4 семестр)

Минск 2021

Список теоретических вопросов

1. Java. Структура программы

2. Java. Классы и интерфейсы

3. Java. Управление памятью. Сборщик мусора

Управление памятью.

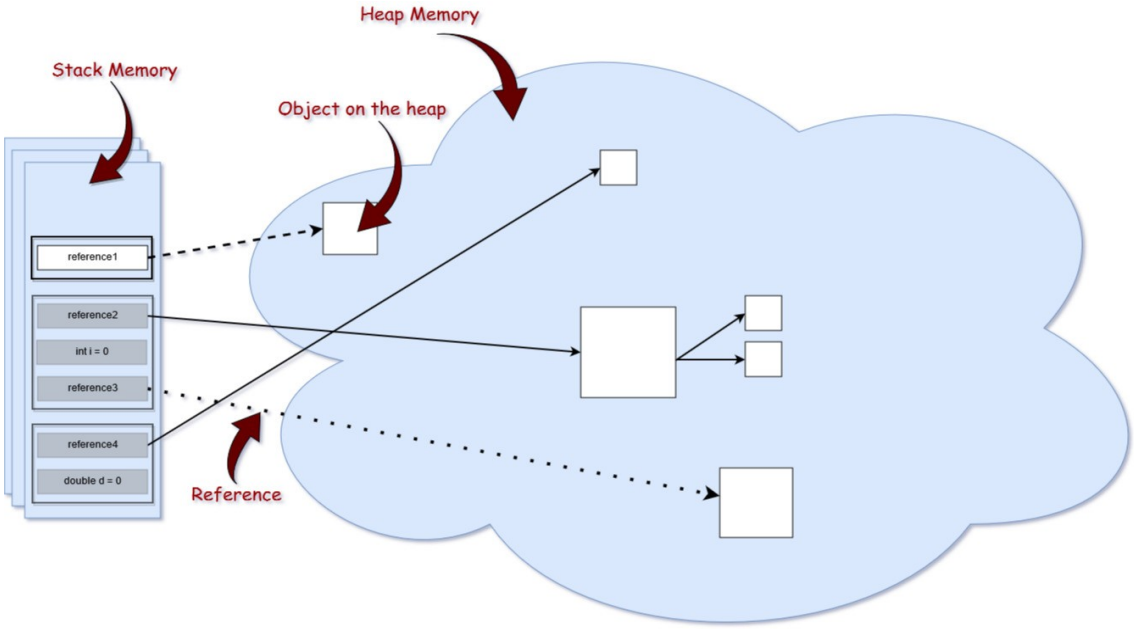
В Java память делится на две большие части:

- Стек (Stack) — отвечает за хранение ссылок на объекты кучи и за хранение примитивных типов, которые содержат само значение, а не ссылку на объект из кучи. Переменные в стеке имеют область видимости (если компилятор выполняет тело метода, он может получить доступ только к объектам из стека, которые находятся внутри тела метода, а когда метод завершается и возвращает значение, верхняя часть стека выталкивается, и активная область видимости изменяется). В Java стековая память выделяется для каждого потока (следовательно, каждый раз, когда поток создается и запускается, он имеет свою собственную стековую память и не может получить доступ к стековой памяти другого потока).
- Куча (Heap) — хранит в памяти фактические объекты, на которые ссылаются переменные из стека.

```
Object obj = new Object();
```

Ключевое слово **new** несет ответственность за обеспечение того, достаточно ли свободного места на куче.

Для каждого запущенного процесса JVM существует только одна область памяти в куче. Следовательно, это общая часть памяти независимо от того, сколько потоков выполняется.



Максимальные размеры стека и кучи не определены заранее - это зависит от работающей JVM машины.

Типы ссылок:

- Сильная ссылка. Это самые популярные ссылочные типы, к которым мы все привыкли. В приведенном выше примере с Object мы фактически храним сильную ссылку на объект из кучи. Объект в куче не удаляется сборщиком мусора, пока на него указывает сильная ссылка или если он явно доступен через цепочку сильных ссылок.
- Слабая ссылка. Попросту говоря, слабая ссылка на объект из кучи, скорее всего, не сохранится после следующего процесса сборки мусора.
- Мягкая ссылка. Эти типы ссылок используются для более чувствительных к памяти сценариев, поскольку они будут собираться сборщиком мусора только тогда, когда вашему приложению не хватает памяти. Следовательно, пока нет критической необходимости в освобождении некоторого места, сборщик мусора не будет касаться легко доступных объектов. В документации Javadocs говорится:

  «все мягкие ссылки на мягко достижимые объекты гарантированно очищены до того, как виртуальная машина выдаст OutOfMemoryError»

- Фантомная ссылка. Используется для планирования посмертных действий по очистке, поскольку мы точно знаем, что объекты больше не живы. Эти типы ссылок считаются предпочтительными для финализаторов.

Ссылки на String.

Ссылки на тип String в Java обрабатываются немного по- другому. Строки неизменяемы, что означает, что каждый раз, когда вы делаете что-то со строкой, в куче фактически создается другой объект. Для строк Java управляет пулом строк (String Pool) в памяти. Это означает, что Java сохраняет и повторно использует строки, когда это возможно.

```
String str1 = «string»;
```

```
String str2 = «string»;
```

В этом случае две ссылки типа String указывают на один и тот же объект в пуле строк, значит при попытке сравнения ссылок, получим true. Однако можно принудить Java создать новый объект в куче при помощи оператора **new**:

```
String str1 = «string»;
```

```
String str2 = new String(«string»);
```

В таком случае имеем два разных объекта в куче с одинаковым значением, и при попытке сравнения ссылок получим false.

Сборщик мусора.

Сборщик мусора (Garbage Collector) — это низкоприоритетный процесс, который запускается периодически и освобождает память, использованную объектами, которые больше не нужны. Разные JVM имеют отличные друг от друга алгоритмы сбора мусора.

Поскольку это довольно сложный процесс и может повлиять на вашу производительность, он реализован разумно. Для этого используется так называемый процесс «Mark and Sweep». Java анализирует переменные из стека и «отмечает» все объекты, которые необходимо поддерживать в рабочем состоянии. Затем все неиспользуемые объекты очищаются.

Так что на самом деле Java не собирает мусор. Фактически, чем больше мусора и чем меньше объектов помечены как живые, тем быстрее идет процесс. Чтобы сделать это еще более оптимизированным, память кучи на самом деле состоит из нескольких частей.

Фактически, JVM имеет три типа сборщиков мусора, и программист может выбрать, какой из них следует использовать. По умолчанию Java выбирает используемый тип сборщика мусора в зависимости от базового оборудования.

1. **Serial GC** (Последовательный сборщик мусора) - однониточный коллектор. В основном относится к небольшим приложениям с небольшим использованием данных.
2. **Parallel GC** (Параллельный сборщик мусора) - даже по названию, разница между последовательным и параллельным будет заключаться в том, что параллельный сборщик мусора использует несколько потоков для выполнения процесса сбора мусора. Этот тип GC также известен как сборщик производительности.
3. **Mostly concurrent GC** (В основном параллельный сборщик мусора). Если вы помните, ранее в этой статье упоминалось, что процесс сбора мусора на самом деле довольно дорогостоящий, и когда он выполняется, все потоки приостанавливаются. Однако у нас есть в основном параллельный тип GC, который утверждает, что он работает одновременно с приложением. Однако есть причина, по которой он «в основном» параллелен. Он не работает на 100% одновременно с приложением. Есть период времени, на который цепочки приостанавливаются. Тем не менее,

пауза делается как можно короче для достижения наилучшей производительности сборщика мусора. На самом деле существует 2 типа в основном параллельных сборщиков мусора:

- **Garbage First** - высокая производительность с разумным временем паузы приложения.
- **Concurrent Mark Sweep** (Параллельное сканирование отметок) - время паузы приложения сведено к минимуму. Начиная с JDK 9, этот тип GC объявлен устаревшим.

Использованные источники.

1. Статья habr.com – Управление памятью Java.
2. Статья javarush.ru – Сборка мусора.

4. Java. Внутренние и анонимные классы. Lambda-выражения

5. Java. Generic

6. Java. Типы потоков ввода-вывода, принцип использования

7. Java. Сериализация объектов в raw

8. Java. Сериализация объектов в XML

9. Java. Чтение и запись в сетевой ресурс

10. Java. Основные возможности библиотеки javax.swing

11. Java. Обработка событий в библиотеке javax.swing

12. Java. Многопоточность в библиотеке javax.swing

13. Java. Enterprise Edition. Библиотека Spring

14. Java. Stream API

15. Java. Reflection API

16. Java. Аннотации

17. Java. Обработка исключений

18. Java. Threads. Создание и управление

19. Java. Threads. Синхронизация

20. Java. Асинхронность

21. Принципы GRASP

GRASP (General Responsibility Assignment Software Patterns – общие шаблоны распределения ответственности) — шаблоны, используемые в объектно-ориентированном проектировании для решения общих задач по назначению ответственности классам и объектам. GRASP паттерны не имеют выраженной структуры, четкой области применения и конкретной решаемой проблемы, а лишь представляют собой обобщенные подходы/рекомендации/принципы, используемые при проектировании дизайна системы.

GRASP состоит из 5 основных и 4 дополнительных шаблонов.

Основные шаблоны.

- **Информационный эксперт (Information Expert).** Шаблон определяет базовый принцип распределения ответственности:

  «*Ответственность должна быть назначена тому, кто владеет максимумом необходимой информации для исполнения — информационному эксперту*»

  Проблема: В системе должна аккумулироваться, рассчитываться и т.п. необходимая информация.

  Решение: Назначить обязанность аккумуляции информации, расчета и т.п. некоему классу (информационному эксперту), обладающему необходимой информацией.

  Рекомендации: Информационным экспертом может быть не один класс, а несколько.

- **Создатель (Creator).** Суть ответственности такого объекта в том, что он создает другие объекты. Есть ряд моментов, которые должны выполняться, когда мы наделяем объект ответственностью создателя:
  - Создатель содержит или агрегирует создаваемые объекты.
  - Создатель использует создаваемые объекты.
  - Создатель знает, как проинициализировать создаваемый объект.
  - Создатель записывает создаваемые объекты.
  - Создатель имеет данные инициализации для объекта.

  Проблема: Кто должен отвечать за создание экземпляров класса?

  Решение: Назначить классу B обязанность создавать объекты другого класса A.

Рекомендации: Логично использовать паттерн если класс B содержит, агрегирует, активно использует и т.п. объекты класса A.

- **Контроллер (Controller).** Контроллер призван решить проблему разделения интерфейса и логики в интерактивном приложении. Это не что иное, как контроллер из MVC парадигмы. Контролер отвечает за обработку запросов и решает кому должен делегировать запросы на выполнение.

  Проблема: Кто должен отвечать за обработку входных системных событий?

  Решение: Обязанности по обработке системных сообщений делегируются специальному классу. Контроллер — это объект, который отвечает за обработку системных событий и не относится к интерфейсу пользователя. Контроллер определяет методы для выполнения системных операций.

  Рекомендации: Для различных прецедентов логично использовать разные контроллеры (контроллеры прецедентов) - контроллеры не должны быть перегружены. Внешний контроллер представляет всю систему целиком, его можно использовать, если он будет не слишком перегруженным (то есть, если существует лишь несколько системных событий).

- **Низкая связанность (Low Coupling).** Если объекты в приложении сильно связанны, то любой их изменение приводит к изменениям во всех связанных объектах. А это неудобно и порождает множество проблем. Low coupling как раз говорит о том что необходимо, чтобы код был слабо связан и зависел только от абстракций.

  Проблема: Обеспечить низкую связанность при создании экземпляра класса и связывании его с другим классом.

  Решение: Распределить обязанности между объектами так, чтобы степень связанности оставалась низкой.

- **Высокое зацепление (High Cohesion).** High Cohesion принцип говорит о том, что класс должен стараться выполнять как можно меньше не специфичных для него задач, и иметь вполне определенную область применения.

  Проблема: Необходимо обеспечить выполнение объектами разнородных функций.

  Решение: Обеспечить распределение обязанностей с высоким зацеплением.

Дополнительные шаблоны.

- **Чистая выдумка (Pure Fabrication).** Существует понятие модели программирования по предметной области, согласно которой, каждой сущности из предметной области соответствует один или более классов программной среды. При этом, обязанности взаимодействия сущностей, как правило накладываются на них самих. Такой подход имеет очевидный недостаток — высокая связность модулей системы. Шаблон Pure Fabrication позволяет решить данную проблему, путем введения в программную среду дополнительного класса (не отражающего реальной сущности из предметной области) и наделение его требуемыми обязанностями.
- **Посредник (Indirection).** Шаблон Indirection реализует низкую связность между классами, путем назначения обязанностей по их взаимодействию дополнительному объекту — посреднику.
- **Полиморфизм (Polymorphism).** Шаблон Polymorphism позволяет обрабатывать альтернативные варианты поведения на основе типа. При этом, альтернативные реализации приводятся к обобщенному интерфейсу.
- **Устойчивость к изменениям (Protected Variations).** Сущность шаблона Protected Variations заключается в устранении точек неустойчивости, путем определения их в качестве интерфейсов и реализации для них различных вариантов поведения.

Использованные источники.

1. Sergey Nemchinskiy — Введение в шаблоны GRASP (видео).
2. Sergey Nemchinskiy — Введение в шаблоны GRASP (презентация).
3. Статья bool.dev – GRASP принципы.
4. Статья habr.com – GRASP паттерны проектирования.
5. Сердюков Р.Е. - GRASP (лекция).

22. Принципы YAGNI, DRY и KISS

Don't Repeat Yourself.

DRY (Don’t Repeat Yourself) – принцип хорошего кода, который гласит:

«*Каждая часть знаний должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы*»

Второе название — SSOT (Single Source Of Truth).

Антипринцип — WET (Write Everything Twice / We Enjoy Typing).

Основные идеи:

- Если дублировать код, то:
  - Придется поддерживать одну и ту же логику и тестировать код сразу в двух местах.
  - Если меняется код в одном месте, его нужно будет изменить и в другом.
- В большинстве случаев дублирование кода происходит из-за незнания системы.

Когда DRY не работает:

- Многопользовательские игры (некоторый вычисления происходят и на сервере и на клиенте, чтобы оптимизировать задержку получения данных с сервера).
- Браузеры (многие браузеры совершают предзагрузку страниц, на которые вероятно пойдет пользователь, чтобы время отклика было меньше).
- Валидация (проверка) данных (на клиенте и сервере).

Keep It Simple, Stupid.

KISS (Keep It Simple, Stupid) – принцип хорошего кода, который гласит:

«Большинство систем работают лучше всего, если они остаются простыми, а не усложняются»

Основные идеи:

- Простые системы будут работать лучше и надежнее.
- Не придумывайте задаче более сложного решения, чем ей требуется.
- Всегда следует искать простой путь, даже если он неочевиден.

- Не стоит перебарщивать с гибкостью и абстракцией в приложении.

Р. Е. Сердюков привел пример:

«Автомат Калашникова — простой механизм, который в общем-то нормально работает»

Правила простого дизайна Кента Бека (Extreme Programming Explained):

- У вас запускаются все тесты.
- Не дублируйте логику. Старайтесь избегать скрытых дубликатов, таких как параллельные иерархии классов.
- Все намерения, важные для программиста, должны быть явно видны.
- Код должен иметь наименьшее возможное количество классов и методов.

You Aren't Gonna Need It.

YAGNI (You Aren't Gonna Need It) – принцип хорошего кода, который гласит:

«*Возможности, которые не описаны в требованиях к системе, просто не должны реализовываться*»

Основные идеи:

- Если пишете код, будьте уверены, что он вам понадобится.
- Не пишите код, если думаете, что он пригодится позже.
- Любые бонусные возможности усложняют сопровождение.
- Если написать ненужный код, то:
  - Тратится время, которое можно было использовать на написание нужного кода.
  - Он (ненужный код) может помешать добавлению нужного функционала в будущем.
  - Он (ненужный код) требует отладки, документирования, тестирования и сопровождения.
  - ПО становиться сложнее.

Р. Е. Сердюков привел пример:

«Надо готовить только то, что вы собираетесь есть. Если вы приготовите больше, оно просто пропадет»

Использованные источники.

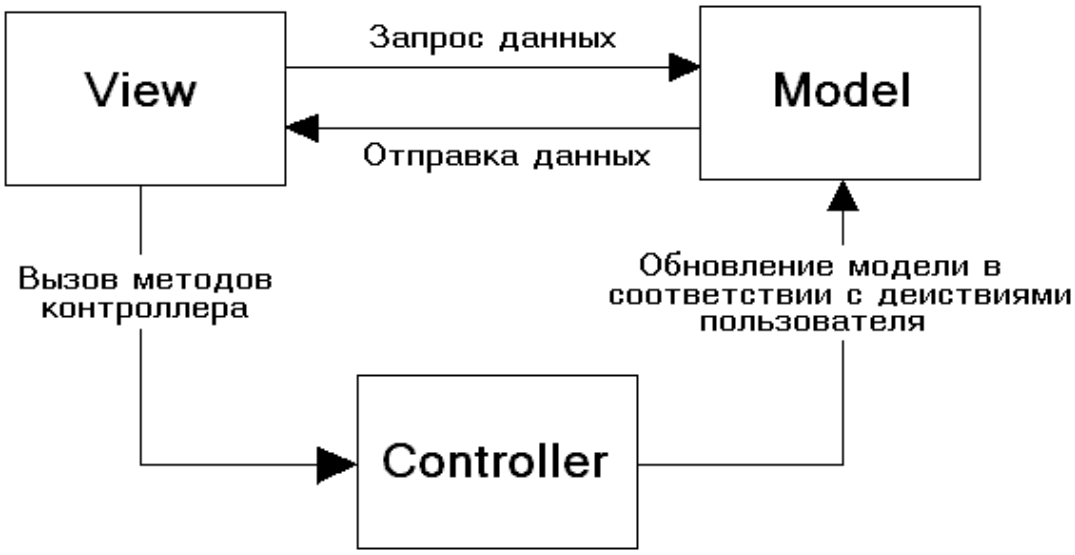
1. Сердюков Р.Е. - YAGNI, DRY, KISS (лекция).

2. Sergey Nemchinskiy — Принцип хорошего кода DRY (видео).
3. Sergey Nemchinskiy — Принцип хорошего кода KISS (видео).
4. Sergey Nemchinskiy — Принцип хорошего кода YAGNI (видео).

23. Операции ввода-вывода. Блокируемый и неблокируемый вызов

24. Паттерн MVC

В графических приложениях для построения интерфейсов пользователя часто применяется тройка классов модель/представление/контроллер (Model/View/Controller — MVC).



MVC состоит из объектов трех видов. *Модель* — это объект приложения, а *представление* — его внешний вид на экране. *Контроллер* описывает, как интерфейс реагирует на управляющие воздействия пользователя. До появления схемы MVC эти объекты в пользовательских интерфейсах смешивались. MVC отделяет их друг от друга, за счет чего повышается гибкость и улучшаются возможности повторного использования.

MVC отделяет представление от модели, устанавливая между ними протокол взаимодействия «подписка/уведомление». Представление должно гарантировать, что внешнее представление отражает состояние модели. При каждом изменении внутренних данных модель уведомляет все зависящие от нее представления, в результате чего представление обновляет себя. Такой подход позволяет присоединить к одной модели несколько представлений, обеспечив тем самым различные представления. Можно создать новое представление, не переписывая модель.

MVC позволяет также изменять реакцию представления на действия пользователя. При этом визуальное воплощение остается прежним. Например, можно изменить реакцию на нажатие клавиши или использовать открывающиеся меню вместо командных клавиш. MVC инкапсулирует механизм определения реакции в объекте *контроллер*. Существует иерархия

классов контроллеров, и это позволяет без труда создать новый контроллер как вариант уже существующего.

Модель.

Под *моделью*, обычно понимается часть содержащая в себе функциональную бизнес-логику приложения. *Модель* должна быть полностью независима от остальных частей продукта. Модельный слой ничего не должен знать об элементах дизайна, и каким образом он будет отображаться. Достигается результат, позволяющий менять представление данных, то как они отображаются, не трогая саму *модель*.

Модель обладает следующими признаками:

- Модель — это бизнес-логика приложения.
- Модель обладает знаниями о себе самой и не знает о контроллерах и представлениях.

Представление.

В обязанности *представления* входит отображение данных полученных от *модели*. Однако, представление не может напрямую влиять на модель. Можно говорить, что представление обладает доступом «только на чтение» к данным.

Представление обладает следующими признаками:

- В представлении реализуется отображение данных, которые получаются от модели любым способом.
- В некоторых случаях, представление может иметь код, который реализует некоторую бизнес-логику.

Примеры представления: HTML-страница, WPF форма, Windows Form.

Контроллер.

Контроллер перехватывает событие извне и в соответствии с заложенной в него логикой, реагирует на это событие изменяя *модель*, посредством вызова соответствующего метода. После изменения *модель* использует событие о том что она изменилась, и все подписанные на это события *представления*, получив его, обращаются к *модели* за обновленными данными, после чего их и отображают.

Признаки контроллера:

- Контроллер определяет, какое представление должно быть отображено в данный момент.
- События представления могут повлиять только на контроллер. Контроллер может повлиять на модель и определить другое представление.
- Возможно несколько представлений только для одного контроллера.

Использованные источники.

1. Паттерны объектно-ориентированного проектирования (книга, глава 1.2).
2. Статья habr.com – Паттерны для новичков: MVC vs MVP vs MVVM.

25. Паттерн MVP и MVVM

26. Паттерн BLOC

27. Внедрение зависимостей

28. Автоматное программирование

29. Реактивное программирование

30. Аспектно-ориентированное программирование