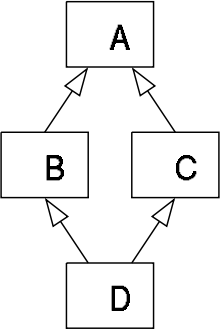
Множественное наследование

Множественное наследование порождает несколько проблем, понимание которых объясняет почему поддержка ООП в Java реализована именно так, а не иначе.

В Java запрещено множественное наследование (посредством ключевого слова extends), но разрешено множественное наследование интерфейсов (реализация посредством ключевого слова implements).

**Ромбовидное наследование**



(англ. *diamond inheritance*) - ситуация в объектно-ориентированных языках программирования с поддержкой множественного наследования, когда два класса B и C наследуют от A, а класс D наследует от обоих классов B и C. При этой схеме наследования может возникнуть неоднозначность: если метод класса D вызывает метод, определенный в классе A (и этот метод не был переопределен в классе D), а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: B или C? Конечно эта проблема решается в современных языках программирования [1].

В Java было реализовано множественное наследование только для интерфейсов, потому что они не содержат реализаций методов, а значит у метода всегда есть только одна реализация, которая и будет вызвана.

Модификаторы в Java

**Модификаторы доступа**

* **private** члены класса доступны только внутри класса
* **package-private** или **default** (по умолчанию) члены класса видны внутри пакета
* **protected** члены класса доступны внутри пакета и в классах-наследниках
* **public** члены класса доступны всем

Во время наследования возможно изменение модификаторов доступа в сторону БОЛЬШЕЙ видимости.

**Модификатор static**

Метод, переменная, блок кода, которые описаны внутри некоторого класса, но вызываются без приложения к конкретному объекту этого класса, называются **статическими**.

У статических сущностей есть такие главные особенности, которые определяют их назначение и использование:

* Статические поля и методы принадлежат не к экземпляру класса, а самому классу. То есть для доступа и работы с ними нет необходимости создавать экземпляр класса.
* Статические поля и методы создаются при загрузке класса;

Статические сущности по поведению близки к глобальным переменным.

Статическими могут быть:

* Внутренние классы (кроме локальных и анонимных классов);
* Переменные;
* Методы;
* Блоки кода. Статические блоки используются для инициализации переменных и выполнения кода один раз при загрузке класса;

**Особенности статических сущностей [3], [4]:**

* Нельзя получить доступ к нестатическим сущностям внутри статического контекста (метода, блока);
* Каждый экземпляр класса имеет одну и ту же копию статической переменной/метода. Из-за этого статические переменные не потокобезопасны;
* Статический метод можно вызвать, используя тип его класса. Поэтому, статические методы подходят в качестве методов-фабрик (factory), и методов-утилит (utility).
* Статические методы не могут быть переопределены (override). Если объявить метод с таким же именем и сигнатурой в классе-наследнике, то метод суперкласса будет "спрятан" вместо переопределения. Это явление известно как сокрытие методов (hiding methods). Вызван будет метод согласно типу переменной.
* Модификатор static также может быть объявлен в статичном блоке, более известным как static initializer block, который будет выполнен во время загрузки класса. Порядок инициализации сверху вниз, в том же порядке, в каком они описаны в исходном коде;
* Статические методы связываются во время компиляции, в отличие от связывания виртуальных или не статических методов, которые связываются во время исполнения на реальном объекте;
* Статические поля не сериализуются;
* Модификатор static import похож на стандартный оператор import, но в отличие от него позволяет импортировать один или все статические члены класса. При импортировании статических методов, к ним можно обращаться как будто они определены в этом же классе;

Параллелизм (Concurrency) в Java

В общем смысле параллелизм – это возможность создания программ в виде нескольких одновременно выполняющихся частей (ветвей).

Параллелизм достигается за счет:

* Вытесняющей многозадачности (когда задачи ставятся в очередь для доступа к ресурсам);
* Истинной многозадачности. Когда в системе есть ресурсы (например, несколько процессоров) для выполнения каждой задачи;

Есть два типа сущностей (единиц выполнения), которые могут выполняться параллельно:

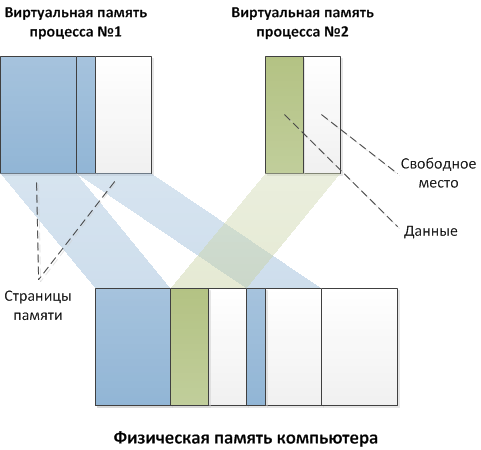
* Процессы
* Потоки

Если в системе несколько процессоров (ядер), то на одно ядро процессора, в каждый момент времени, приходится одна единица исполнения. За распределение единиц выполнения по процессорам отвечает операционная система и JVM.

Когда речь идет о concurrency в Java, то чаще всего подразумевается несколько потоков выполняющихся одновременно.

**Процессы**

Процесс - это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из одного процесса. Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств).

Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память. Страницы виртуальной памяти могут проецироваться на физическую память в произвольном порядке.  
  
Схема этого взаимодействия представлена на картинке.  
  
  
При запуске программы операционная система создает процесс, загружая в его адресное пространство код и данные программы, а затем запускает главный поток созданного процесса [5].

Java умеет работать с процессами. Процессом может быть родное приложение ОС или еще один экземпляр JVM (в случае Java-приложения).

Для управления процессами используются классы java.lang.Process и java.lang.ProcessBuilder. Существуют также разные библиотеки для работы с процессами, например, Apache Commons.

**Потоки**

Поток – это единица исполнения кода. Можно еще сказать, что поток – это путь выполнения программы. Каждый поток последовательно выполняет инструкции внутри процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

**Многопоточность в Java** – это возможность выполнения двух или более разделов программы (потоков) в одно и то же время в рамках одного процесса (программы).

С логической точки зрения, многопоточность означает, что несколько строк из одной и той же программы могут быть выполнены в одно и то же время.

Каждый процесс имеет хотя бы один выполняющийся поток. Тот поток, с которого начинается выполнение программы, называется **главным потоком**. В Java, после создания процесса, выполнение главного потока начинается с метода main().

Потоки могут находиться в одном из следующих состояний:

* **Состояние Новый (New)**. После создания экземпляра потока, он находится в состоянии Новый до тех пор, пока не вызван метод start(). В этом состоянии поток не считается живым;
* **Состояние Работоспособный (Runnable)**. Поток переходит в состояние Работоспособный, когда вызывается метод start(). Поток может перейти в это состояние также из состояния Работающий или из состояния Блокирован. Когда поток находится в этом состоянии, он считается живым;
* **Состояние Работающий (Running)**. Поток переходит из состояния Работоспособный в состояние Работающий, когда Планировщик потоков выбирает его как работающий в данный момент;
* **Состояние Живой, но не работоспособный (Alive, but not runnable)**. Поток может быть живым, но не работоспособным по нескольким причинам. Он может быть в состояниях Ожидания, Сна или Блокировки;
* **Ожидание (Waiting)**. Поток переходит в состояние Ожидания, вызывая метод wait(). Вызов notify() или notifyAll() может перевести поток из состояния Ожидания в состояние Работоспособный. Метод sleep() переводит поток в состояние Сна на заданный промежуток времени в миллисекундах;
* **Блокировка (Blocked)**. Поток может перейти в это состояние, в ожидании ресурса, такого как ввод/вывод или из-за блокировки другого объекта. В этом случае поток переходит в состояние Работоспособный, когда ресурс становится доступен;
* **Мёртвый (Dead)**. Поток считается мёртвым, когда его метод run() полностью выполнен. Мёртвый поток не может перейти ни в какое другое состояние, даже если для него вызван метод start().

**Завершение процесса и демоны**

В Java процесс завершается тогда, когда завершается последний его поток. Даже если метод main() уже завершился, но еще выполняются порожденные им потоки, система будет ждать их завершения.

Однако это правило не относится к особому виду потоков - демонам. Если завершился последний обычный поток процесса, и остались только потоки-демоны, то они будут принудительно завершены и выполнение процесса закончится. Чаще всего потоки-демоны используются для выполнения фоновых задач, обслуживающих процесс в течение его жизни [5].

**Класс Thread и интерфейс Runnable**

Многопоточность в JavaCore поддерживается с помощью класса Thread, его методов и дополняющего его интерфейса Runnable. Thread инкапсулирует поток выполнения.

Чтобы создать новый поток, программа должна:

* Либо расширить класс Thread переопределив его метод run();
* Либо реализовать интерфейс Runnable и передать Runnable-объект объекту класса Thread в конструкторе;

Класс Thread определяет несколько методов, которые помогают управлять потоками.

|  |  |
| --- | --- |
| **Метод** | **Назначение** |
| getName | Получить имя потока. |
| getPriority | Получить приоритет потока. |
| isAlive | Определить, выполняется ли поток. |
| join | Ожидать завершения потока. |
| run | Входная точка потока. |
| sleep | Приостановить выполнение потока на заданное время. |
| start | Запустить поток вызовом его метода. |

В Java существуют средства для принудительного завершения потока:

* Thread.stop() завершает поток незамедлительно после своего выполнения;
* Thread.suspend(), приостанавливает поток;
* Thread.resume(), продолжает выполнение потока;

Эти методы объявлены устаревшими и их использование крайне нежелательно.

Вместо принудительного завершения потока применяется схема, в которой каждый поток сам ответственен за своё завершение. Поток может остановиться либо тогда, когда он закончит выполнение метода run(), (main() — для главного потока) либо по сигналу из другого потока.

Причем как реагировать на такой сигнал - дело, самого потока. Получив его, поток может выполнить некоторые операции и завершить выполнение, а может его проигнорировать и продолжить выполняться.

Описание реакции на сигнал завершения потока реализуется программистом.

**Рекомендации по использованию**

Класс который наследовал Thread уже не сможет наследовать другой класс. Используйте Thread, только если классу нужно просто выполниться как отдельному потоку.

Интерфейс Runnable может быть реализован любым классом, что не лишает этот класс возможности наследования.

**Thread dump**

Thread dump – это описание всех потоков и их ресурсов, которые активны в данный момент на JVM. Thread dump – это способ узнать что каждый поток делал в конкретный момент времени.

Thread dump содержит такую информацию:

* Имя потока;
* Тип потока и его приоритет;
* Java Thread ID. Получается из java.lang.Thread.getId();
* Native Thread ID. Родной идентификатор, присвоенный ОС;
* Java Thread State and detail. Позволяет быстро узнать состояние потока;
* Java Thread Stack Trace. Стек вызовов потока. Наиболее важная часть дампа, анализ которой позволяет определить причину неполадок;
* Java Heap breakdown. Дамп памяти кучи.

**Как сгенерировать thread dump**

Thread dump можно получить такими способами:

1. С помощью **jstack**

* Узнать PID вашего Java-процесса. Способ зависит от ОС;
* Установить и использовать утилиту jstack. Входит в JDK начиная с версии 1.5;

1. Unix, Linux, Mac OSX, JDK <= 1.4

* Использовать команду kill –QUIT;

1. Windows JDK 1.x

* Использовать утилиту javadump.exe
* Использовать утилиту jconsole

1. Если выполнение идет на серверах приложений, использовать инструментарий этих серверов.
2. Task20
3. <http://www.quizful.net/post/features-of-the-application-of-modifiers-in-java>
4. <http://info.javarush.ru/translation/2014/04/15/10-%D0%B7%D0%B0%D0%BC%D0%B5%D1%82%D0%BE%D0%BA-%D0%BE-%D0%BC%D0%BE%D0%B4%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%82%D0%BE%D1%80%D0%B5-Static-%D0%B2-Java.html>
5. <http://samolisov.blogspot.com/2007/11/java.html>
6. <http://habrahabr.ru/post/164487/>