**Java: Управление ресурсами.**

Как я заметил, многие, начинающие программировать на Java, не вполне чётко представляют себе, как следует управлять ресурсами на этом языке. В данной статье я попытался изложить своё видение на этот аспект программирования.

**Что может, а чего не может GC.**

Одним из особенностей языка Java является автоматическая сборка мусора - пресловутый Garbage Collector (сокращённо - GC). GC - это фоновый процесс, который занимается поиском и освобождением неиспользуемых объектов. Много копий было сломано в дискуссиях, хорошо это или плохо, причём существенные аргументы есть как у сторонников, так и у противников этой технологии. Но я не об этом. Примем существование GC как данность и рассмотрим особенности программирования в данной среде.

Одним из следствий наличия GC (и отсутствия размещения объектов в стеке) является отсутствие деструкторов, аналогичным существующих в C++ - автоматически вызываемых методов для уничтожения объектов по выходу из области видимости. Вместо них есть другие механизмы, которые позволяют добиться того же самого.

Другим следствием является необходимость следить за ссылками на объекты, потому как GC именно по наличию ссылки на объект определяет его нужность. Для возможности управления строгостью ссылок в JDK 1.2 был введён новый пакет java.lang.ref, классы которого позволяет создавать так называемые "слабые" ссылки на объекты. Это особенно полезно для реализации кэширования объектов.

Начитавшись рекламных статей, может показаться, что очисткой ресурсов вообще не стоит заниматься - всё сделает GC. Тут важно понимать, что в понятие ресурсов входит не только оперативная память, но так же и многое другое. Например - открытые файлы, сокеты, ResultSet-ы и Statement-ы в JDBC, и т.д. С очисткой памяти GC в последних JVM справляется довольно неплохо, но вот освобождение других ресурсов, как правило, нельзя делать асинхронно. В противном случае легко получить так называемые "плавающие" ошибки, которые то появляются, то исчезают.

Над разработкой алгоритма работы GC довольно долго корпели разработчики, так что иногда его поведение может показаться несколько странным. В частности, он может освобождать объекты в произвольном порядке, никак не связанном с порядком их создания или очистки последних ссылок на них. Так же, для примера, в JDK 1.3, GC начинает освобождать объекты только тогда, когда текущий пул памяти заполняется. При этом одновременно, этот пул расширяется, пока не дойдёт до заданного верхнего лимита. Одновременно учитывается время последнего доступа и размер объекта. И т.д. и т.п.

По умолчанию верхний лимит пула зависит от системы. В JDK 1.3 он равен 64 мегабайтам. В некоторых случаях лучше снизить эту планку, особенно когда на машине памяти не очень много - тем самым программа быстрей дойдёт до этого лимита и начнёт использовать память повторно.

Большинство JVM не отдают обратно системе память, которую они забрали под пул. Я знаю только одну JVM, которая это делает - это Microsoft JVM. Все остальные хапают до указанного лимита.

**Деструкторов нет. А что вместо?**

Многие программисты (особенно только что пересевшие с C++), видят замену деструкторов в виде методов finalize. Это не есть правильно. Во первых, этот метод не вызовется, пока на объект имеются ссылки. Нет гарантии, что он вызовется тогда, когда вам это нужно. Он может вообще никогда не вызваться. Есть, конечно, метод System.runFinalizersOnExit(), но он уже давно deprecated из-за возможных deadlock-ов. Как справедливо заметил Bruce Eckel в замечательной книжке Thinking in Java метод finalize стоит использовать только для того, чтобы освободить память (или другой, не очень критичный ресурс), выделенную в native методах. Для прочих же ресурсов следует добавить явный вызов метода очистки ресурсов (он может называться по-разному - close, dispose, cleanup, и т.д.), внутри конструкции finally. Внутри finally - для того, чтобы не прозевать какое-нибудь исключение. Пример типичных конструкций:

OutputStream out = new FileOutputStream(...);

try

{

...

}

finally { out.close(); }

PreparedStatement stmt = ...;

try

{

stmt.setInt(1,...);

ResultSet rs = stmt.executeQuery();

try

{

...

}

finally { rs.close(); }

}

finally { stmt.close(); }

Если Вы будете писать программы в таком стиле, то многие проблемы обойдут Вас стороной - поверьте, я уже наступал на эти грабли. :-)

**Memory Leak в Java и как с ним бороться.**

Казалось бы - какие могут быть Memory Leak (утечки памяти), когда есть GC. Но, тем не менее, они возможны. Эти утечки отличаются по своему происхождению от утечек в C/C++ (таких в Java вообще быть не может, за исключением возможных ошибок в JVM, но этот вариант мы здесь не рассматриваем). Они возникают из-за забытых ссылок.

Например - в GUI часто используется механизм слушателей (Listener-ов). Это когда один объект регистрируется в другом объекте для обратного вызова по наступлению какого-нибудь события. Проблема тут может быть одна - если Вы забудете разрегистрировать объект, когда он становится ненужным. При этом он останется болтаться в списке, ну и GC, разумеется, не будет удалять этот объект, пока не придёт время удалить объект со списком (а этого может и не случиться до завершения программы).

В результате - список будет пухнуть, пухнуть, программа работает всё медленней и медленней, и, в конце концов, вылетает OutOfMemoryError.

Отследить такой сценарий можно при помощи специальных программ - профайлеров. Самые известные на данный момент - JProbe и OptimizeIt.

Бороться с этими вещами довольно просто - надо не забыть разрегистрироваться. Если Вы уверены, что объект-слушатель удаляется вместе с объектом - источником событий (например, когда они оба сидят внутри диалога), то этого делать не нужно - GC сам с ними разберётся.

**Кэширование объектов.**

Кэширование объектов применяется для повышения производительности. Когда объект меняется редко (или вообще не меняется в процессе работы программы), то часто полезно прочитать его один раз, и запомнить на него ссылку. Но ссылка - это противоречие между алгоритмом работы GC, а значит, этот объект никогда не будет собран GC (даже если он больше не нужен). Иногда это большого значения не имеет (памяти много, или все объекты требуются очень часто), но чаще желательно всё-таки иметь возможность почистить неиспользуемые объекты.

В JDK 1.2 для этих целей были добавлены несколько классов, самые интересные из них - WeakReference и SoftReference. Weak ("слабая") ссылка будет очищена, если на объект нет более "сильных" ссылок и объект попался под руку GC. Soft ("мягкая") ссылка будет очищена, если на объект нет обычных ссылок, и, к объекту не доступались по этой ссылке определённое время (эвристически рассчитываемое в зависимости от текущего окружения). Для кэширования лучше всего подходят ссылки типа Soft, т.к. в них учитывается частота обращения к объекту.

Непосредственно пользоваться этими ссылками не очень удобно, лучше написать контейнер с их использованием. Вот, например реализация класса SoftHashtable:

import java.lang.ref.SoftReference;

import java.util.Hashtable;

public class SoftHashtable extends Hashtable

{

public SoftHashtable() {}

public Object put(Object key,Object obj)

{

SoftReference ref = (SoftReference)super.put(key,new SoftReference(obj));

if( ref==null ) return null;

return ref.get();

}

public Object get(Object key)

{

SoftReference ref = (SoftReference)super.get(key);

if( ref==null ) return null;

return ref.get();

}

public Object remove(Object key)

{

SoftReference ref = (SoftReference)super.remove(key);

if( ref==null ) return null;

return ref.get();

}

}

Этот класс является неполным, при желании Вы можете перекрыть остальные методы Hashtable. С помощью этого класса Вы уже можете легко организовать свой кэш объектов:

private static Hashtable objCache = new SoftHashtable();

public SuperObject getObject(String key)

{

SuperObject o = (SuperObject)objCache.get(key);

if( o==null )

{

o = loadObject(key);

objCache.put(key,o);

}

return o;

}

Таким образом, если GC обнаружит, что памяти осталось мало, а в кэше болтаются объекты, к которым давно никто не обращался, он сможет их почистить.