Память и Garbage collector в Java

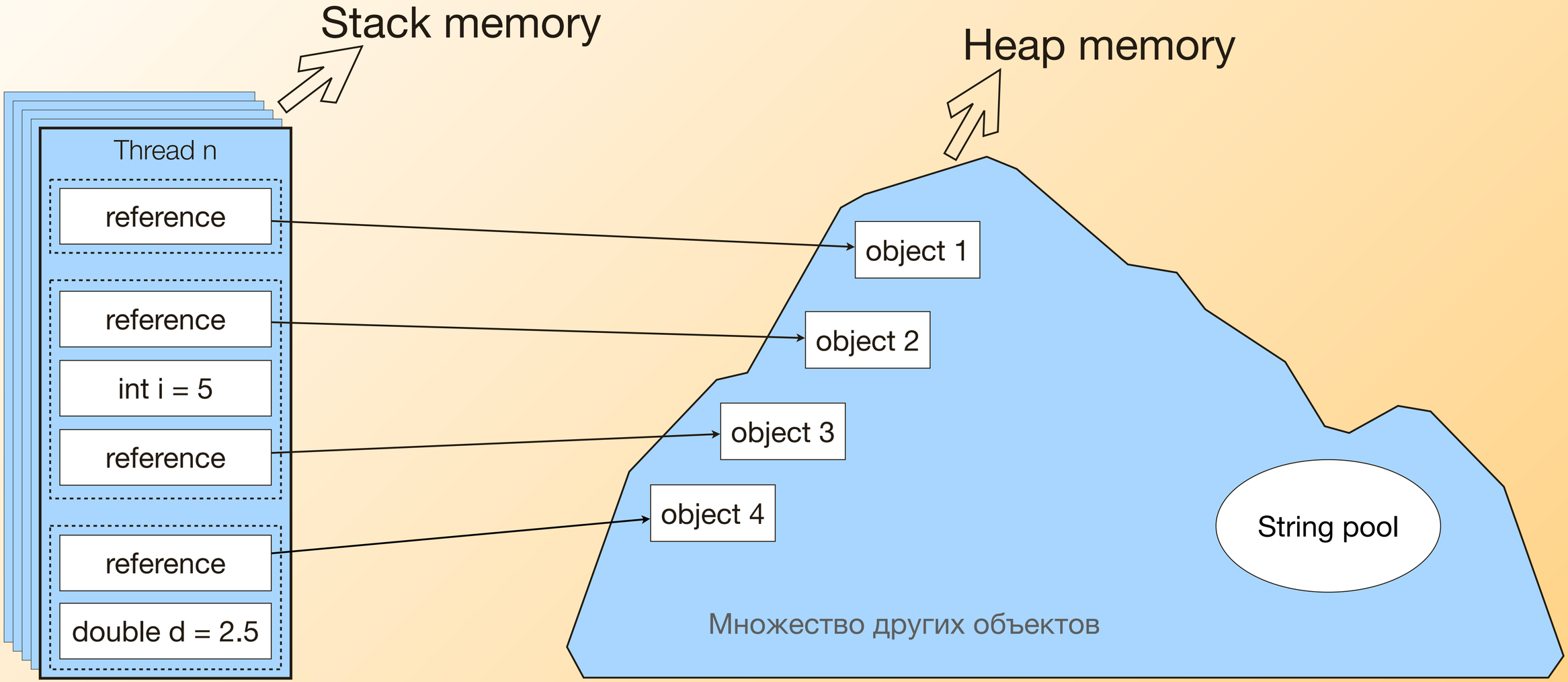
Память

В Java память приложения можно разделить на **стек**, (Stack memory), и **кучу**, (Heap memory).

В куче хранятся объекты.

В стеке хранятся ссылки на объекты из кучи и переменные примитивных типов данных.

Память


Stack memory
Heap memory
Thread n
reference
reference
int i = 5
reference
reference
double d = 2.5
object 1
object 2
object 3
object 4
String pool
Множество других объектов

Stack memory

- работает по принципу стека, (удивительно, но факт);
- хранит ссылки на объекты;
- хранит примитивы;
- переменные имеют область видимости, (невозможно получить доступ к переменным, объявленным в методе A, из метода B);
- для каждого потока выполнения отдельная stack memory.

Heap memory

- здесь фактически хранятся объекты, ссылки на которые лежат в stack memory;
- одна для всех потоков.

String pool

- отдельная область Heap Memory, в которой хранятся строки;
- хранит все строки, созданные во время работы приложения, следовательно при повторном создании строки, значение которой уже существовало, новый объект не создастся, мы просто получим еще одну ссылку на уже существующий объект.

Вообще не всегда, строка может быть удалена сборщиком мусора.

String pool

```
String s1 = "hello";
String s2 = "hello";
```

В данном примере **один** «hello» создастся в String pool, а s1 и s2, оба будут ссылаться на один этот «hello».

```
String s3 = new String("hello");
String s4 = new String("hello");
```

В данном примере **два** объекта со значением «hello» создадутся в Heap memory. И s3 будет ссылаться на один из объектов, а s4 на второй.

Числовые пулы

Также существуют числовые пулы для: Byte, Short, Integer, Long, Character.

Но в отличие от String pool, это не отдельный раздел памяти в Heap, а механизм кэширования.

При старте JVM создается массив объектов соответствующего типа определенного диапазона.

Числовые пулы

	Диапазон	Обращение	Расширяемость
Byte pool	-128 … 127	Byte b = 3; Byte b = valueOf(5);	Нет
Short pool	-128 … 127	Short s = 3; Short s = valueOf(5);	Нет
Integer pool	-128 … 127	Integer i = 3; Integer i = valueOf(5);	Нет
Long pool	-128 … 127	Long l = 3; Long l = valueOf(5);	Да, (-XX:AutoBoxCacheMax=n)*
Character pool	0 … 127, (ASCII)	Character c = 3; Character c = valueOf(5);	Нет

** При расширении Integer pool двигается только верхняя граница, нижняя всегда равна -128*

Взаимодействие с памятью

Во время работы приложение создает и хранит в памяти какие-либо данные, (в случае ООП языка можно сказать «объекты»). Возникает вопрос: а что делать, когда она закончится?

Все языки программирования можно разделить на две группы:

- **unmanaged** - ЯП, которые сами управляют памятью. Например: C, C++, Delphi;

- **managed** - ЯП, которые отдает необходимость управлять памятью на откуп программисту. Например: Java, C#, Python, JavaScript, Ruby, Go, PHP, Swift.

Взаимодействие с памятью

С **unmanaged ЯП** все понятно: сам объект создал, попользовался им, удалил из памяти, когда необходимость в нем пропала. То есть у нас полностью ручное управление, и все зависит исключительно от места, из которого растут руки программиста.

А как работают **managed ЯП**, при работе с которыми программист почти ни на что не влияет?

Garbage collector

Можно очень условно обобщить названия механизмов управления памятью в различных ЯП до Garbage collector, (сборщик мусора).

Garbage collector - процесс, освобождающий память приложения путем уничтожения неиспользуемых объектов.

У GC всего две задачи: найти мусор, отчистить мусор.

Мусор - это объект в памяти, к которому из программного кода больше невозможно получить доступ.

Garbage collector

Существует несколько реализаций GC, работающих по различным алгоритмам, каждый из которых по своему решает проблему отслеживания и уничтожения уже ненужных объектов. Какие-то реализации лучше работают на больших размерах heap-а, какие-то лучше работают на средних размерах и меньше.

При этом по спецификации нет никаких правил для реализации GC, кроме сбора объектов, которые гарантировано более не могут быть использованы.

Поиск мусора

Нахождение мусора на самом деле не тривиальная задача.

Существуют 2 подхода для поиска мусора:

- reference counting;
- tracing.

Reference counting

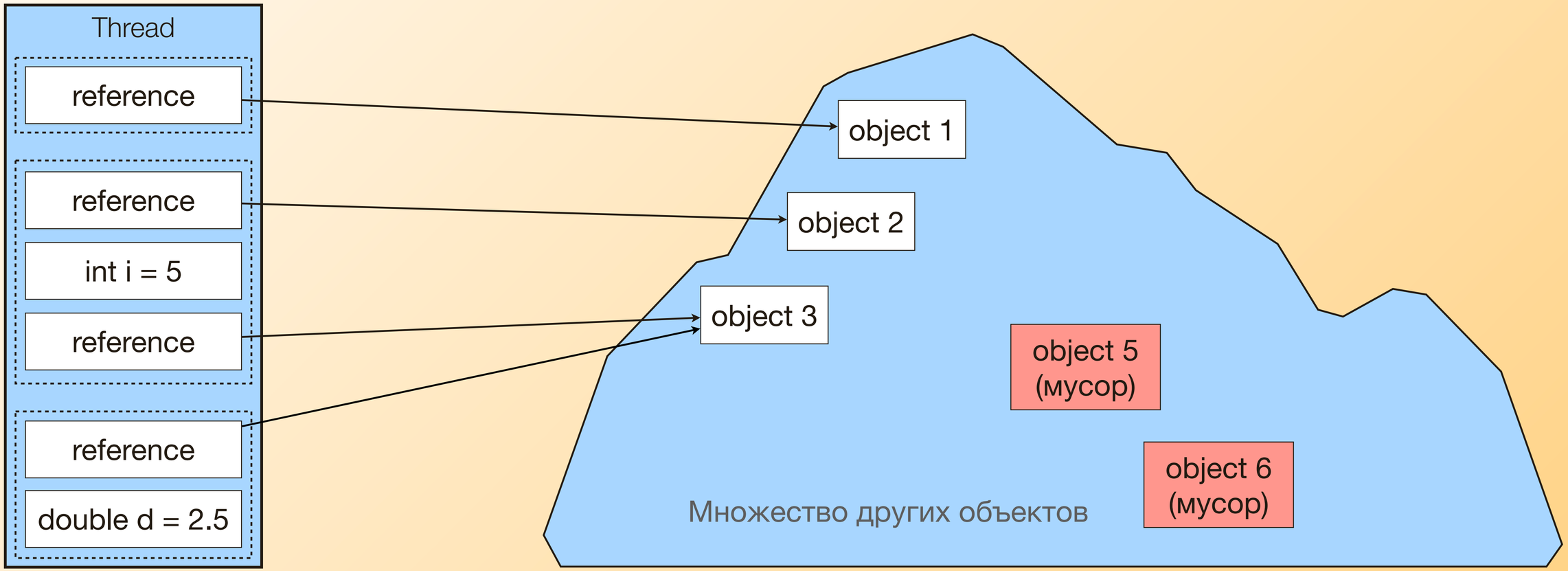
Из названия понятно, что подход основан на подсчете ссылок.

Каждый объект имеет счетчик, который хранит информацию, сколько ссылок указывают на этот объект.

При создании ссылки на объект, счетчик увеличивается. Когда ссылка уничтожается, счетчик уменьшается.

Если значение счетчика равно нулю, объект считается мусором и память, которую он занимает, можно отчищать.

Reference counting



Reference counting

Рассмотрим данный подход на примере объекта Integer(500).

На 2 строке объявляется ссылка i на Integer(500), а его счетчик равен 1.

На 3 строке объявляется ссылка j на тот же объект, теперь его счетчик равен 2.

```
1 public void doSomething() {
2   Integer i = 500;
3   Integer j = i;
4   Integer sum = i + j;
5   j = 400;
6 }
```

На 5 строке ссылка j переписывается на объект Integer(400). А счетчик Integer(500) уменьшается и равен 1.

По выходу из метода локальные ссылки i и j уничтожаются, так как они живут только внутри метода. Счетчик на объект Integer(500) уменьшается и равен 0.

Теперь Integer(500) - мусор, и GC может его удалить.

Reference counting

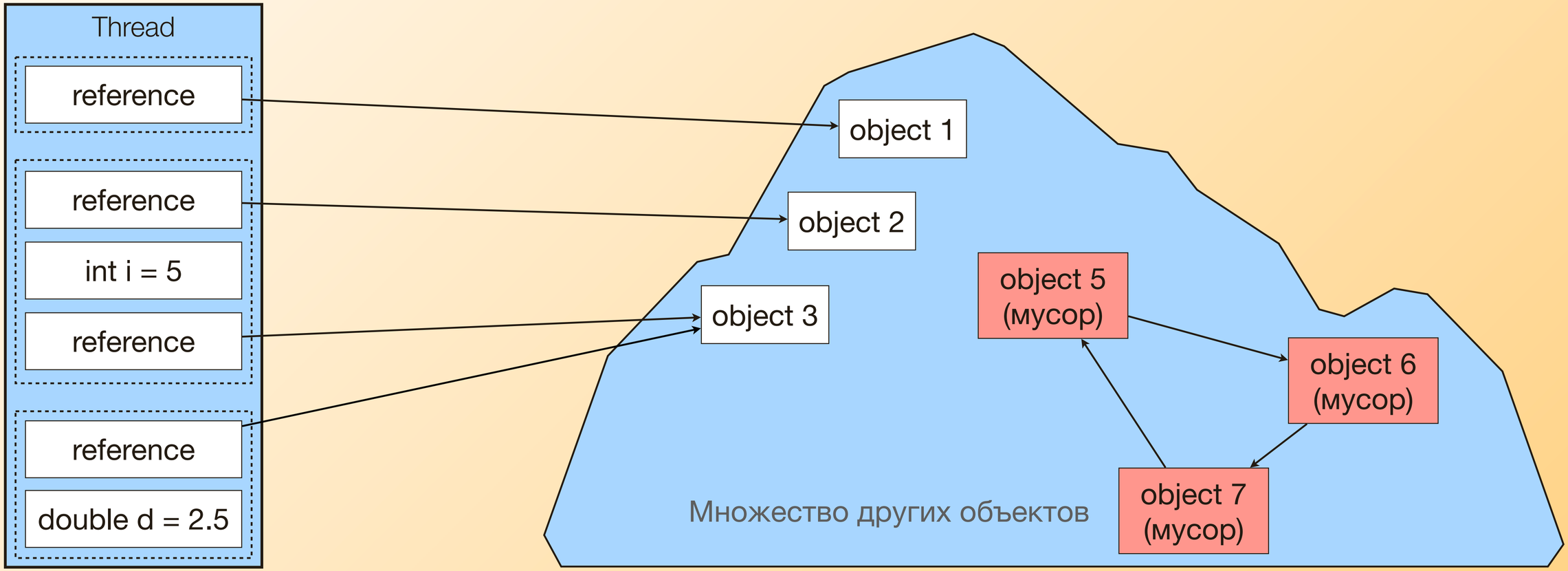
Плюсы:

- простота реализации;
- отсутствуют длительные паузы работы приложения для поиска мусора.

Минусы:

- не обнаружит циклические ссылки;
- требуется дополнительное время на изменение счетчика;
- требуется дополнительная память на счетчик;
- проблемы с многопоточностью.

Циклические ссылки



Tracing

Идея в следующем:

Если до объекта можно дойти от «корня», то это живой объект. До чего дойти не можем - мусор.

GC Root

Этот корень называется GC Root.

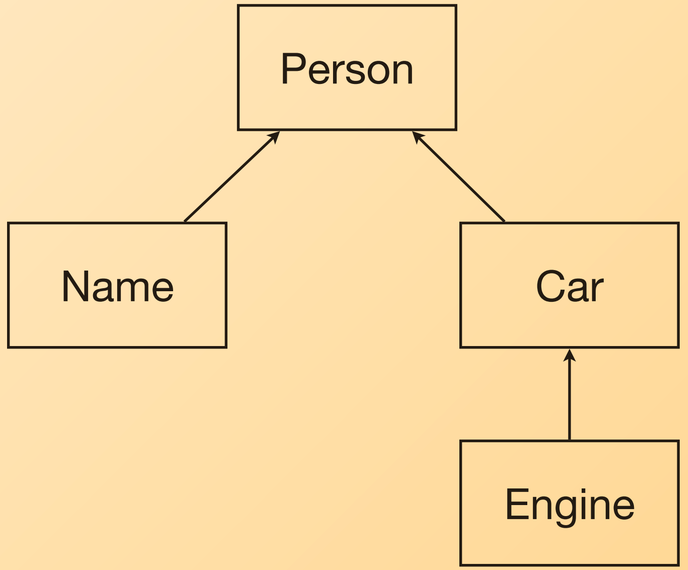
Что может являться GC Root?

- параметры методов;
- локальные переменные;
- потоки Java;
- статические переменные;
- ссылки из JNI.

JNI - Java Native Interface - стандартный механизм для запуска кода под управлением виртуальной машины Java, который написан на unmanaged-языках и скомпонован в виде динамических библиотек.

GC Root

```
1 Person p = new Person();
2 p.setName("John");
3 p.setCar(new Car());
4 p.getCar().setEngine(new Engine());
```



В данной ситуации Person - GC root.

Допустим, Person живой объект, и следовательно Name, Car, Engine тоже живые объекты, т.к. мы можем добраться до них от Person.

GC Root

Даже в таком простом коде у нас есть следующие GC Root:

- поток, выполняющий метод main;
- параметры метода main.

```
1 public class Main {
2  public static void main(String[] args) {
3    System.out.println("Hello, World!");
4  }
5 }
```

Также GC Root могли быть локальные переменные метода main и статические переменные класса Main.

Очистка мусора

Очистка мусора процесс очень понятный, но сложный. И в этот раз из-за сложности в нашем мире есть несколько алгоритмов, выполняющих задачу очистки.

Существуют 3 подхода для поиска мусора:

- копирующая сборка;
- отслеживание и очистка, (Mark-and-Sweep);
- отслеживание и очистка с дефрагментацией, (Mark-and-Sweep Compact);
- разделение на поколения.

Остановка мира, (Stop the world)

Stop the world - это полная остановка выполнения приложения для запуска цикла сборки мусора.

Преимущества:

- проще определять достижимость объектов, так как граф объектов заморожен;
- проще перемещать объекты в heap memory, так как в момент остановки мы можем перевести память в некорректное состояние.

Если пауза критична, то одну большую паузу превращают в много маленьких, нагружая этим процессор.

Копирующая сборка мусора

Память условно делится на две области: from-space и to-space.

1. Все объекты создаются в from-space, и from-space полностью.
2. Приложение останавливается, (Stop the world).
3. Все живые объекты перемещаются в to-space.
4. Все мертвые объекты очищаются.
5. From-space и to-space меняются местами.

Mark-and-Sweep

1. Все объекты создаются в памяти и она заполняется.
2. Приложение останавливается, (Stop the world).
3. Помечаем все живые объекты, (Mark).
4. Чистим все мертвые объекты, (Sweep).
5. Снимаем все пометки с живых объектов.

Mark-and-Sweep Compact

1. Все объекты создаются в памяти и она заполняется.
2. Помечаем все мертвые объекты, (Mark).
3. Приложение останавливается, (Stop the world).
4. Чистим все мертвые объекты, (Sweep).
5. Дефрагментируем память, (Compact).

Сборка мусора на поколениях

Ввиду того, что любой из предыдущих алгоритмов плохо работает в каких-то ситуациях, каждый из них можно назвать плохим.

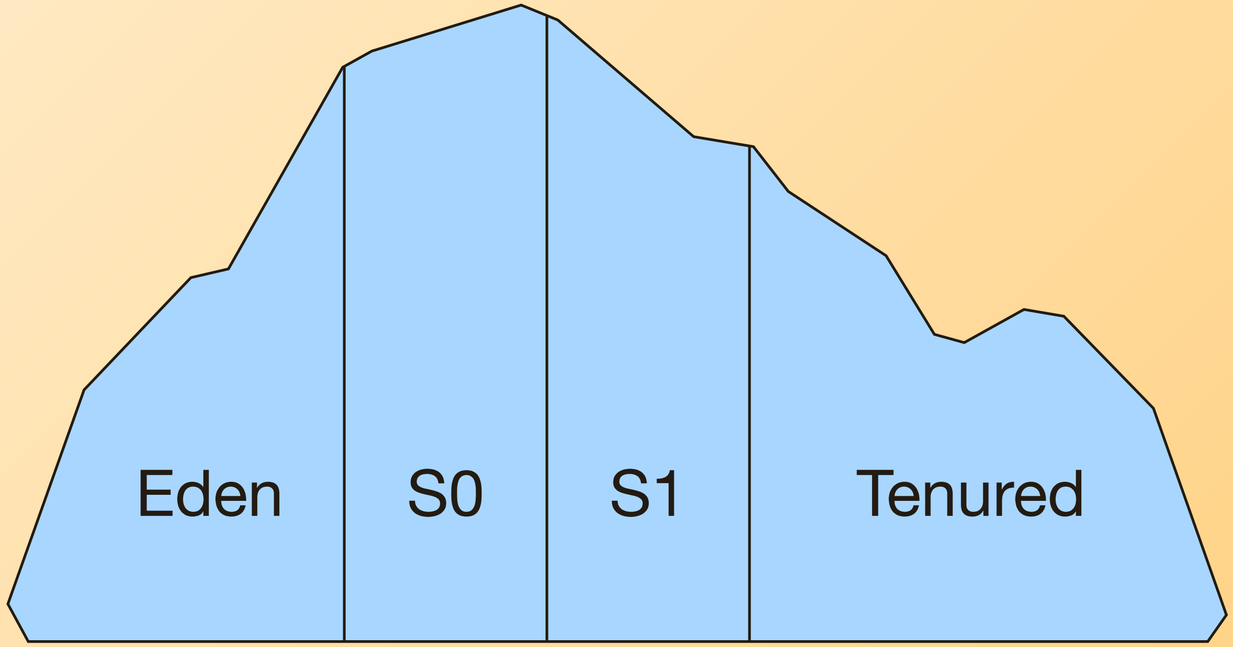
И исходя из этого был создан новый на основе «слабой гипотезы о поколениях», состоящей из двух закономерностей.

1. Большинство объектов живут либо очень долго, либо очень недолго, а также долгоживущих объектов очень мало.
2. Связей между старыми и новыми объектами очень мало.

Сборка мусора на поколениях

В этом алгоритме память условно делится на две области:

- young generation, состоящий из трех разделов: Eden, S0 и S1, (survivor space);
- old generation, (Tenured).



Сборка мусора на поколениях

Исходя из разделения на области памяти, есть разделения и на сборки мусора:

- minor, собирает мусор в Eden, S0 и S1;
- major, собирает мусор в Tenured;
- full, собирает мусор везде.

Сборка мусора на поколениях

Алгоритм выглядит следующим образом:

1. Объекты создается в Eden.
2. При запуске minor - сборки все живое из Eden перемещается в S0.
3. При запуске следующей minor - сборки все живое из Eden и S0 перемещается в S1, и, грубо говоря, S1 и S0 меняются местами.
4. После нескольких пережитых minor - сборок все живое из S0 перемещается в Tenured.

Сравнение алгоритомов очистки

	Копирующая сборка	Mark and Sweep	Mark and Sweep Compact	Разделение на поколения
Использование памяти	Половина	Вся	Вся	Вся
Фрагментация	Нет	Есть	Нет	Или есть, или нет
Производительность	Зависит от числа живых объектов	Зависит от размера всей кучи	Mark and Sweep + компактификация	Высокая
Подходит для маложивущих объектов	Хорошо	Плохо	Плохо	Отлично
Подходит для долгоживущих объектов	Плохо	Хорошо	Хорошо	Отлично
Остановка приложения	Полная	Полная	Полная	Частичная
Пауза в работе	Быстрая, если мало живых	Быстрая на маленьких кучах	Долгая	Минимальная
Сложность реализации	Простой	Сложнее	Еще Сложнее	Самый сложный

Реализации GC

Вот несколько реализаций GC:

- Serial GC;
- Parallel GC;
- CMS GC;
- G1 GC;
- ZGC;
- Shenandoah GC.

Serial GC

Однопоточный сборщик.

Работает на основе поколений, (в old generation использует Mark and Sweep Compact).

Подходит для однопоточных приложений с малой нагрузкой.

Простой и экономичный, но медленный.

Аргумент JVM для использования последовательного сборщика мусора ***-XX:+UseSerialGC****.*

Parallel GC

Тоже самое, что Serial GC, но работает многопоточно.

Используется по умолчанию.

CMS GC

Удален в Java 17, (deprecated в Java 14).

Работает по принципу Mark and Sweep, для сокращения длительности пауз, путем увеличения их количества.

Аргумент JVM для использования последовательного сборщика мусора ***-XX:+UseConcMarkSweepGC****.*

G1 GC

Замена CMS GC.

Работает на основе поколений, но делит их на регионы, (от 1 Мб до 32 Мб).

По поколениям работает аналогично Serial GC, но для каждой сборки мусора выбирает самые «грязные» регионы.

Аргумент JVM для использования последовательного сборщика мусора ***-XX:+UseG1GC****.*

ZGC

«Промышленный» GC.

Работает с кучами до 16 Тб.

Отличная производительность на больших кучах.

Требует больших ресурсов CPU.

Существует с Java 11, стабильный с Java 16.

Аргумент JVM для использования последовательного сборщика мусора ***-XX:+UseZGC****.*

Shenandoah GC

«Промышленный» GC.

Работает с кучами до 2 Тб.

Отличная производительность на маленьких кучах.

Требует больших ресурсов CPU.

Существует с Java 12, стабильный с Java 15.

Аргумент JVM для использования последовательного сборщика мусора ***-XX:+UseG1GC****.*

Типы ссылок. StrongReference

Самый популярный ссылочный тип.

Объект созданный таким образом не будет удален сборщиком мусора:

- если на него указывает сильная ссылка
- если он явно не доступен через цепочку сильных ссылок.

```
StringBuilder builder = new StringBuilder();
```

Типы ссылок. WeakReference

Типы ссылок. SoftReference

```
StringBuilder builder = new StringBuilder();
```

Типы ссылок. PhantomReference

```
StringBuilder builder = new StringBuilder();
```