1. **Object.**
2. **Khái quát chung**

* **Khái niệm:** Trong Java, object là một thực thể của một lớp (class), được tạo ra từ class. Mỗi object có các thuộc tính (properties) và phương thức (methods) được xác định bởi class của nó. Nói cách khác, object là một sự triển khai cụ thể của một class, chứa các dữ liệu và các hành vi mà class đã định nghĩa.
* **Tầm quan trọng:** Object là nền tảng của mọi thao tác trong lập trình hướng đối tượng. Các thao tác xử lý dữ liệu và hành vi đều diễn ra trên object. Điều này giúp lập trình viên xây dựng các ứng dụng phức tạp một cách rõ ràng, dễ bảo trì và dễ phát triển.

1. **Object được lưu như thế nào trong Java?**

* **Khai báo:**

- Để khai báo một đối tượng thuộc lớp, ta sử dụng cú pháp:

**<Tên lớp> [Tên đối tượng];**

**Ví dụ: School highSchool;**

- Chúng ta sử dụng toán tử new và gọi đến hàm khởi tạo (constructor)

của lớp để khởi tạo đối tượng. Hàm constructor có tên trùng với tên của lớp.

**<Tên lớp> [Tên đối tượng] = new <Tên lớp>;**

**Ví dụ: smartPhone iphone = new smartPhone();**

* **Quá trình lưu Object:** Trong Java, khi một object được tạo ra, nó sẽ được lưu trong vùng nhớ heap (heap memory), nơi dành riêng cho các object được cấp phát động (dynamic memory allocation). Java sử dụng một cơ chế tự động gọi là Garbage Collection để quản lý bộ nhớ của các object trong heap, đảm bảo rằng bộ nhớ sẽ được thu hồi khi object không còn được sử dụng.
* **Note: Vùng nhớ Heap:**

- **Khái niệm**: Là vùng nhớ chính được sử dụng để lưu trữ các object và được quản lý bởi Garbage Collector. Tất cả object được tạo bằng từ khóa new đều lưu trên heap.

- **Phân loại:**

+ **Young Generation:** Nơi lưu trữ các object mới được tạo. Các object thường xuyên được tạo và thu hồi trong giai đoạn này.

+ **Old Generation (Tenured Generation):** Các object tồn tại lâu (sống sót qua nhiều lần thu hồi trong Young Generation) sẽ được chuyển vào đây.

+ **Permanent Generation (PermGen) hoặc Metaspace:** Lưu trữ thông tin về các class và metadata, không phải là các object thông thường. (Lưu ý: từ Java 8, PermGen đã được thay thế bởi Metaspace).

1. **Wrapper Class.**
2. **Khái niệm.**

* **Wrapper Class** trong Java là các lớp bao bọc (wrap) cho các kiểu dữ liệu nguyên thủy (primitive types), như int, double, boolean, v.v., biến chúng thành các đối tượng (objects). Java cung cấp một lớp bao bọc tương ứng cho mỗi kiểu dữ liệu nguyên thủy để hỗ trợ xử lý dữ liệu ở dạng đối tượng trong các cấu trúc và API yêu cầu object thay vì primitive.

1. **Tác dụng.**

Wrapper Class giúp:

* **Đại diện kiểu nguyên thủy dưới dạng đối tượng:** Một số cấu trúc dữ liệu như ArrayList chỉ chấp nhận đối tượng, không chấp nhận kiểu nguyên thủy. Wrapper Class cho phép bạn đưa các giá trị nguyên thủy vào những cấu trúc này.
* **Cung cấp các phương thức tiện ích:** Wrapper Class có các phương thức hữu ích để xử lý dữ liệu, chẳng hạn như chuyển đổi giữa chuỗi và giá trị nguyên thủy, kiểm tra giá trị lớn nhất/nhỏ nhất của kiểu dữ liệu, so sánh giá trị, v.v.
* **Hỗ trợ tính năng tự động Boxing/Unboxing:** Java hỗ trợ chuyển đổi tự động giữa các kiểu dữ liệu nguyên thủy và Wrapper Class tương ứng giúp mã dễ đọc và ngắn gọn.

1. **Phân loại.**

Có 8 wrapper class tương ứng với 8 kiểu dữ liệu nguyên thủy trong Java như sau:

| **Kiểu nguyên thủy** | **Wrapper class** |
| --- | --- |
| boolean | Boolean |
| char | Character |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

* Để có thể dùng các phương thức được chuẩn bị sẵn trong wrapper class cho một giá trị thuộc kiểu nguyên thủy, trước hết chúng ta cần phải tạo ra một đối tượng từ wrapper class để chứa giá trị đó.
* Để tạo ra đối tượng từ wrapper class **(boxing)** trong Java, chúng ta sử dụng tới phương thức thành viên valueOf() trong các class wrapper, với cú pháp sau đây:

**WrapperClassName name = WrapperClassName.valueOf(value);**

* Sau khi đã gán giá trị nguyên thủy vào đối tượng của wrapper class tương ứng, chúng ta có thể lấy ra giá trị **(unboxing)** này nhiều lần trong chương trình Java, thông qua các phương thức tương ứng trong các class.

1. **Auto boxing và Auto unboxing.**

**Auto-boxing và auto-unboxing** là cơ chế tự động chuyển đổi giữa các kiểu dữ liệu nguyên thủy (primitive types) và các lớp bao bọc (Wrapper Class) tương ứng trong Java. Đây là một tính năng được giới thiệu từ Java 5, giúp mã dễ đọc và dễ viết hơn bằng cách loại bỏ sự cần thiết phải chuyển đổi thủ công giữa các kiểu nguyên thủy và đối tượng của chúng.

**1. Auto-boxing**

* Auto-boxing là quá trình tự động chuyển đổi từ một kiểu dữ liệu nguyên thủy (primitive) thành một đối tượng của Wrapper Class tương ứng.
* Khi bạn gán một giá trị nguyên thủy cho một biến thuộc Wrapper Class, Java tự động "bọc" (box) giá trị đó thành một đối tượng.

**Ví dụ:**

int x = 5;

Integer obj = x; // Auto-boxing, chuyển int thành Integer

Trong ví dụ trên, giá trị 5 kiểu int được tự động "bọc" vào đối tượng kiểu Integer. Thao tác này tương đương với:

Integer obj = Integer.valueOf(x);

**2. Auto-unboxing**

* Auto-unboxing là quá trình tự động chuyển đổi từ một đối tượng của Wrapper Class về kiểu dữ liệu nguyên thủy tương ứng.
* Khi bạn gán một biến kiểu Wrapper Class vào một biến kiểu nguyên thủy hoặc thực hiện các phép tính, Java sẽ tự động "gỡ bọc" (unbox) giá trị của đối tượng.

**Ví dụ:**

Integer obj = 10;

int y = obj; // Auto-unboxing, chuyển Integer thành int

Trong ví dụ này, Java sẽ tự động chuyển đổi obj từ Integer thành int. Thao tác này tương đương với:

int y = obj.intValue();

**3. Cách Java thực hiện Auto-boxing và Auto-unboxing**

* Khi thực hiện auto-boxing, Java sử dụng các phương thức valueOf() của Wrapper Class để chuyển đổi kiểu nguyên thủy thành đối tượng. Ví dụ: Integer.valueOf(int), Double.valueOf(double).
* Khi thực hiện auto-unboxing, Java sử dụng các phương thức như intValue(), doubleValue(), v.v., để chuyển đổi đối tượng về kiểu nguyên thủy tương ứng.

**4. Ví dụ về Auto-boxing và Auto-unboxing trong Biểu thức**

* Auto-boxing và auto-unboxing rất hữu ích khi làm việc với các biểu thức hỗn hợp giữa kiểu nguyên thủy và Wrapper Class:

Integer a = 100;

Integer b = 200;

int sum = a + b; // Java tự động unbox a và b thành kiểu int để thực hiện phép cộng

System.out.println(sum); // Kết quả là 300

* Trong ví dụ trên: Java tự động "gỡ bọc" (unbox) a và b từ kiểu Integer về kiểu int. Thực hiện phép cộng hai giá trị int, rồi gán vào biến sum.

**5. Ứng dụng của Auto-boxing và Auto-unboxing**

* Trong Collections: Các cấu trúc dữ liệu như ArrayList, HashMap, v.v., yêu cầu đối tượng thay vì kiểu nguyên thủy. Auto-boxing cho phép sử dụng trực tiếp các kiểu nguyên thủy trong những cấu trúc này.

ArrayList<Integer> list = new ArrayList<>();

list.add(10); // Auto-boxing 10 thành Integer trước khi thêm vào list

int num = list.get(0); // Auto-unboxing từ Integer thành int khi lấy giá trị ra

**6. Một số lưu ý khi sử dụng Auto-boxing và Auto-unboxing**

* Hiệu suất: Quá trình boxing và unboxing liên tục có thể gây ra hiệu suất kém trong các ứng dụng yêu cầu tính toán nhiều. Vì mỗi lần boxing/unboxing đều tạo ra một đối tượng mới trên heap, nó sẽ làm tăng tải cho Garbage Collector.
* NullPointerException: Khi một biến thuộc Wrapper Class có giá trị null mà bị unbox, sẽ gây ra lỗi NullPointerException.

Integer obj = null;

int num = obj; // Lỗi NullPointerException vì obj là null

1. String và StringBuilder.
2. String

a, Khái quát chung.

* Thông thường, string là một chuỗi các ký tự. Nhưng, trong java string là một đối tượng biểu diễn một nối tiếp của các ký tự. Lớp java.lang.String được sử dụng để tạo đối tượng string.
* Lớp String trong java cung cấp rất nhiều các phương thức để thực hiện các thao tác với chuỗi như: compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring(), ...
* Lớp java.lang.String được implements từ các interface Serializable, Comparable and CharSequence.
* String là bất biến (immutable) tức là không thể thay đổi. Có nghĩa là khi nào bạn thay đổi giá trị của bất kỳ chuỗi nào thì một instance mới được tạo ra. Đối với chuỗi có thể thay đổi, bạn có thể sử dụng các lớp StringBuffer và StringBuilder

b, Cách khai báo đối tượng String.

Có 2 cách để khai báo đối tượng String:

* Sử dụng String Literal

- Để làm cho Java sử dụng bộ nhớ hiệu quả hơn (Vì nếu chuỗi đã tồn tại trong Pool thì sẽ không có đối tượng mới được tạo ra).

- String literal được tạo ra bằng cách sử dụng 2 dấu nháy kép.

**Ví dụ:**

String s = “ProPTIT”;

- Mỗi khi bạo tạo một biến string literal, đầu tiên JVM sẽ kiểm tra xem giá trị đó đã tồn tại trong Pool chưa. Nếu chuỗi này đã tồn tại trong Pool, thì giá trị của biến sẽ được tham chiếu đến instance đã được tạo ra trong Pool. Nếu chuỗi này không tồn tại trong Pool, một instance mới được tạo ra và đặt vào trong Pool.

**Ví dụ:**

String s1 = "welcome";

String s2 = "welcome"; // se khong tao instance moi

Theo ví dụ trên, chỉ có một đối tượng chuỗi "Webcome" được tạo ra.

**Note: Các đối tượng String được lưu trong một vùng nhớ đặc biệt đó là Pool hằng số chuỗi.**

* Sử dụng từ khóa new

- String được tạo ra bằng cách sử dụng từ khóa new theo cú pháp:

**String <tên đối tượng> = new String(“<chuỗi>”);**

**Ví dụ:**

String s = new String("Welcome");

- Trong trường hợp này, JVM sẽ tạo ra một đối tượng string mới như một đối tượng trong bộ nhớ HEAP và chữ "Welcome" sẽ được đặt trong Pool. Biến s sẽ tham chiếu tới đối tượng được tạo ra trong HEAP.

c, Các phương thức của lớp String trong java

* Lớp java.lang.String cung cấp nhiều phương thức hữu ích để thực hiện các thao tác trên chuỗi của các giá trị char.

1. StringBuilder.

* Trong java, lớp StringBuilder được sử dụng để tạo chuỗi có thể thay đổi (mutable). Lớp StringBuilder trong java tương tự như lớp StringBuilder ngoại trừ nó không đồng bộ(non-synchronized).

a, Các Constructor quan trọng của lớp StringBuilder trong java

- **StringBuilder():** Tạo ra một Builder chuỗi với dung lượng ban đầu là 16.

- **StringBuilder(String str):** Tạo ra một Builder chuỗi với chuỗi cụ thể.

- **StringBuilder(int capacity):** Tạo ra một Builder chuỗi với dung lượng được chỉ định như độ dài chuỗi.

b, Các phương thức của StringBuilder

* **public StringBuilder append(String s):** được sử dụng để nối thêm các chuỗi được chỉ định với chuỗi này. Các phương thức append() được nạp chồng như append(char), append(boolean), append(int), append(float), append(double),…

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **public** **class** StringBuilderExam1 {  **public** **static** **void** main(String args[]) {    StringBuilder sb = **new** StringBuilder("Hello ");    sb.append("Java");//đến đây chuỗi ban đầu đã bị thay đổi    System.out.println(sb);//in Hello Java   }  } |

* **public StringBuilder insert(int offset, String s):** được sử dụng để chèn chuỗi chỉ định với chuỗi này tại vị trí quy định. Các phương thức insert() được nạp chồng như insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double), …

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **public** **class** StringBuilderExam2 {  **public** **static** **void** main(String args[]) {    StringBuilder sb = **new** StringBuilder("Hello ");    sb.insert(1, "Java");//đến đây chuỗi ban đầu đã bị thay đổi    System.out.println(sb);//in -> HJavaello   }  } |

* **public StringBuilder replace(int startIndex, int endIndex, String str):** được sử dụng để thay thế chuỗi từ vị trị startIndex đến endIndex bằng chuỗi str.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **public** **class** StringBuilderExam3 {  **public** **static** **void** main(String args[]) {    StringBuilder sb = **new** StringBuilder("Hello");    sb.replace(1, 3, "Java");    System.out.println(sb);//in -> HJavalo   }  } |

* **public StringBuilder delete(int startIndex, int endIndex):** được sử dụng để xóa chuỗi từ vị trí startIndex đến endIndex.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **public** **class** StringBuilderExam4 {  **public** **static** **void** main(String args[]) {    StringBuilder sb = **new** StringBuilder("Hello");    sb.delete(1, 3);    System.out.println(sb);//in -> Hlo   }  } |

* **public StringBuilder reverse():** được sử dụng để đảo ngược chuỗi.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **public** **class** StringBuilderExam5 {  **public** **static** **void** main(String args[]) {    StringBuilder sb = **new** StringBuilder("Hello");    sb.reverse();    System.out.println(sb);//in -> olleH   }  } |

* **public int capacity():** được sử dụng để trả về dung lượng hiện tại.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **public** **class** StringBuilderExam6 {  **public** **static** **void** main(String args[]) {    StringBuilder sb = **new** StringBuilder();    System.out.println(sb.capacity());//mặc định là 16    sb.append("Hello");    System.out.println(sb.capacity());//đến đây vẫn là 16    sb.append("java is my favourite language");    System.out.println(sb.capacity());//đến đây là (16\*2)+2=34 i.e (dung lượng cũ\*2)+2   }  } |

* **public void ensureCapacity(int minimumCapacity):** được sử dụng để đảm bảo dung lượng ít nhất bằng mức tối thiểu nhất định.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **public** **class** StringBuilderExam7 {  **public** **static** **void** main(String args[]) {    StringBuilder sb = **new** StringBuilder();    System.out.println(sb.capacity());//mặc định là 16    sb.append("Hello");    System.out.println(sb.capacity());//đến đây là 16    sb.append("java is my favourite language");    System.out.println(sb.capacity());//đến đây là (16\*2)+2=34 i.e (dung lượng cũ\*2)+2    sb.ensureCapacity(10);//đến đây không có sự thay đổi    System.out.println(sb.capacity());//đến đây là 34    sb.ensureCapacity(50);//đến đây là (34\*2)+2    System.out.println(sb.capacity());//đến đây là 70   }  } |

* **public char charAt(int index):** được sử dụng trả về ký tự tại vị trí quy định.

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7 | **public** **class** CharAtExample {  **public** **static** **void** main(String args[]) {    String name = "hello java";  **char** ch = name.charAt(4);    System.out.println(ch);   }  } | |  |

* **public int length():** được sử dụng trả về chiều dài của chuỗi nghĩa là tổng số ký tự.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **public** **class** CharAtExample {  **public** **static** **void** main(String args[]) {    String name = "hello java";  **char** ch = name.length();    System.out.println(ch);   }  } |

* **public String substring(int beginIndex):** được sử dụng trả về chuỗi con bắt đầu từ vị trí được chỉ định.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **public** **class** SubstringExample {  **public** **static** **void** main(String args[]) {    String s1 = "hellojava";    System.out.println(s1.substring(3));// "lojava"   }  } |

* **public String substring(int beginIndex, int endIndex):** được sử dụng trả về chuỗi con với vị trí bắt đầu và vị trí kết thúc được chỉ định.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **public** **class** SubstringExample {  **public** **static** **void** main(String args[]) {    String s1 = "hellojava";    System.out.println(s1.substring(3, 7));// "loja"    System.out.println(s1.substring(3));// "lojava"   }  } |

1. Equals và hashcode, toán tử ==
2. Equals.

* Phương thức equals() so sánh hai chuỗi đưa ra dựa trên nội dung của chuỗi. Nếu hai chuỗi khác nhau nó trả về false. Nếu hai chuỗi bằng nhau nó trả về true.
* Phương thức equals() của lớp String được ghi đè từ phương thức equals() của lớp Object.
* Cú pháp: **<Object 1>.equals(<Object 2>)**

1. Toán tử ==

* Phương thức equals() được thiết kế để so sánh hai đối tượng về mặt ngữ nghĩa (bằng cách so sánh các thành viên dữ liệu của lớp), trong khi toán tử == so sánh hai đối tượng về mặt kỹ thuật (bằng cách so sánh các tham chiếu của chúng, nghĩa là địa chỉ bộ nhớ).

Ví dụ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **package** vn.viettuts;    **public** **class** EqualExample1 {  **public** **static** **void** main(String[] args) {          String s1 = **new** String("This is a string");          String s2 = **new** String("This is a string");            System.out.println("s1 == s2: " + (s1 == s2));          System.out.println("s1.equals(s2): " + (s1.equals(s2)));      }  } |

Ở ví dụ trên, so sánh tham chiếu (toán tử ==) trả về false vì s1 và s2 là hai đối tượng khác nhau được lưu trữ ở các vị trí khác nhau trong bộ nhớ. Trong khi so sánh ngữ nghĩa trả về true bởi vì s1 và s2 có cùng giá trị (“This is a string”) có thể được coi là bằng nhau về mặt ngữ nghĩa.

1. Hashcode

* Định nghĩa phương thức hashCode() trong lớp Object:

**public native int hashCode();**

* Số băm này được sử dụng bởi các collection dựa trên bảng băm như Hashtable , HashSet và HashMap để lưu trữ các đối tượng trong các container nhỏ được gọi là "nhóm". Mỗi nhóm được liên kết với mã băm và mỗi nhóm chỉ chứa các đối tượng có mã băm giống hệt nhau.
* Nói cách khác, một bảng băm nhóm các phần tử của nó bằng các giá trị mã băm của chúng. Sự sắp xếp này giúp cho bảng băm định vị một phần tử một cách nhanh chóng và hiệu quả bằng cách tìm kiếm trên các phần nhỏ của collection thay vì toàn bộ collection.
* Dưới đây là các bước để định vị một phần tử trong một bảng băm:

1. Nhận giá trị mã băm của phần tử được chỉ định bằng cách gọi phương thức hashCode().
2. Tìm nhóm thích hợp được liên kết với mã băm đó.
3. Bên trong nhóm, tìm phần tử chính xác bằng cách so sánh phần tử được chỉ định với tất cả các phần tử trong nhóm. Bằng phương thức equals() của phần tử đã chỉ định được gọi.

* Khi chúng ta thêm các đối tượng của một lớp vào một collection dựa trên bảng băm (HashSet, HashMap ), phương thức hashCode() của lớp được gọi để tạo ra một số nguyên (có thể là một giá trị tùy ý). Con số này được sử dụng bởi bộ sưu tập để lưu trữ và định vị các đối tượng một cách nhanh chóng và hiệu quả, vì collection dựa trên bảng băm không duy trì thứ tự các phần tử của nó.

**Note:** Việc thực thi phương thức mặc định hashCode() trong lớp Object trả về một số nguyên là địa chỉ bộ nhớ của đối tượng. Bạn nên ghi đè phương thức trong các lớp của bạn. Hầu hết các lớp trong JDK ghi đè phương thức hashCode() của riêng chúng, chẳng hạn như String , Date , Integer , Double , v.v.

* Các quy tắc cho phương thức equals() và hashCode() trong Java

Như đã giải thích ở trên, collection dựa trên bảng băm xác định một phần tử bằng cách gọi phương thức hashCode() và equals() của nó, vì vậy khi ghi đè các phương thức này chúng ta phải tuân theo các quy tắc sau:

1. Khi phương thức equals() được ghi đè, phương thức hashCode() cũng phải được ghi đè.
2. Nếu hai đối tượng bằng nhau, mã băm của chúng phải bằng nhau.
3. Nếu hai đối tượng không bằng nhau, không có ràng buộc về mã băm của chúng (mã băm của chúng có thể bằng nhau hay không).
4. Nếu hai đối tượng có mã băm giống nhau, thì không có ràng buộc nào về sự bình nhau của chúng (chúng có thể bằng nhau hay không).
5. Nếu hai đối tượng có mã băm khác nhau, chúng không được bằng nhau.

=> Nếu chúng ta vi phạm các quy tắc này, các collection sẽ hoạt động có thể không đúng như các đối tượng không thể tìm thấy, hoặc các đối tượng sai được trả về thay vì các đối tượng chính xác.

1. Truyền tham chiếu và truyền tham trị

* Trong Java, khi ta truyền tham số vào một phương thức (method), Java sử dụng cách truyền theo giá trị (pass by value). Điều này có nghĩa là Java tạo một bản sao của giá trị và truyền bản sao đó vào phương thức, chứ không phải tham chiếu đến đối tượng gốc.
* Trong Java, có 2 cách để truyền tham số cho phương thức:

1. Tham trị (pass by value)
2. Tham chiếu (pass by reference)
3. Truyền tham trị trong Java

* Trong Java, khi gọi một phương thức và truyền một giá trị cho phương thức, được gọi là truyền tham trị. Việc thay đổi giá trị chỉ có hiệu lực trong phương thức được gọi, không có hiệu lực bên ngoài phương thức.
* Trong Java, truyền tham trị dành cho các tham số có kiểu dữ liệu nguyên thủy là byte, short, int, long, float, double, boolean, char.

1. Truyền tham chiếu trong Java

* Trong Java, khi gọi một phương thức và truyền một tham chiếu cho phương thức, được gọi là truyền tham chiếu. Việc thay đổi giá trị của biến tham chiếu bên trong phương thức làm thay đổi giá trị của nó.
* Trong Java, tất các phương thức có tham số là biến có kiểu là các lớp (class) đều là kiểu tham chiếu.

1. Tại sao String thay đổi được mặc dù pass by value?

* String trong Java là immutable (bất biến), nghĩa là khi thay đổi nội dung của một String, Java tạo ra một đối tượng mới trong bộ nhớ thay vì thay đổi đối tượng ban đầu.
* Nếu ta thay đổi tham chiếu String bên trong hàm (gán một giá trị String khác cho nó), Java chỉ thay đổi bản sao của tham chiếu đó, còn tham chiếu ban đầu bên ngoài hàm sẽ không bị ảnh hưởng.

1. Garbage Collectors
2. Khái niệm

* Garbage Collectors (dưới đây sẽ gọi tắt là GC) được định nghĩa như là một quá trình tự động thực thi nhiệm vụ quản lý bộ nhớ. Trong quá trình chạy chương trình, các đối tượng được tạo ở vùng nhớ heap, một phần bộ nhớ dành cho chương trình. Sau cùng, sẽ có một vài đối tượng mà chương trình không cần dùng đến. Các đối tượng này sẽ được garbage collector truy tìm và xóa bỏ để thu hồi lại dung lượng bộ nhớ. Khác biệt rất nhiều khi chúng ta làm việc với C/C++, việc quản lý bộ nhớ phải thực hiện thủ công, còn GC sẽ thực hiện tự động.

1. Cách hoạt động

* Đầu tiên chúng ta hãy nhớ lại kiến trúc JVM, trong module "Run time area" chúng ta có 2 vùng nhớ là vùng nhớ Stack (dùng để lưu trữ tham số và các biến local) và vùng nhớ Heap (dùng để lưu trữ các đối tượng sau khi khởi tạo bằng từ khóa new và các biến static). Bộ nhớ Heap là bộ nhớ cần dọn dẹp nhất vì các Object không cần dùng nữa phải được xóa bỏ để giải phóng bộ nhớ.
* Có nhiều tiến trình thu gom rác khác nhau nhưng phổ biến nhất Oracle HotSpot. Mặc dù HotSpot có nhiều tiến trình thu gom rác được tối ưu cho từng trường hợp khác nhau nhưng tất cả đều theo một phương thức cơ bản nhất. Đầu tiên, các Object không được tham chiếu sẽ được đánh dấu sẵn sàng để được dọn rác. Ở bước thứ hai, trình thu gom rác sẽ tiến hành xóa các Object đó. Ở bước thứ ba, vùng nhớ của các Object còn lại sẽ được nén lại và nằm liền kề nhau trong bộ nhớ Heap. Quá trình này sẽ giúp việc cấp phát bộ nhớ cho Object mới dễ dàng hơn.
* Ở vùng nhớ Heap được chia làm 3 vùng nhớ nhỏ hơn, tạm gọi là Young generation, Old generation và Permanent generation.

1. Young generation: được chia thành nhiều vùng nhớ nhỏ hơn là Eden(khởi tạo) và Survivor(sống sót). Các Object vừa được khởi tạo sẽ nằm ở vùng Eden, sau chu kì quét đầu tiên nếu Object đó còn tồn tại thì sẽ được chuyển sang vùng Survivor. Tại đây cái Object được GC theo dõi liên tục, nếu như qua nhiều chu kì quét mà Object vẫn còn được sử dụng thì lúc này Object sẽ được chuyển sang vùng nhớ thứ thứ hai.
2. Old generation: là nơi chứa những Object tòn tại đủ "lâu", còn "lâu" như thế nào thì lại tùy thuộc vào thuật toán của từng bộ GC.
3. Permanent generation: không chứa Object mà đây là nơi để chứa các metadata của JVM như các class và method của ứng dụng. Do đó nếu các class và method không còn được sử dụng nữa thì GC sẽ coi chúng là "rác" và dọn dẹp nó.
4. Ưu - nhược điểm

* Ưu điểm: Nó làm cho việc sử dụng bộ nhớ java hiệu quả bởi vì bộ thu gom rác (Garbage Collection) loại bỏ các đối tượng không được tham chiếu từ bộ nhớ heap một cách tự động.
* Nhược điểm: Ảnh hưởng tới tốc độ, hiệu suất của chương trình, cùng một output nhưng chương trình C++ sẽ cho ra kết quả nhanh hơn Java.