06_01_ch3_발표자료

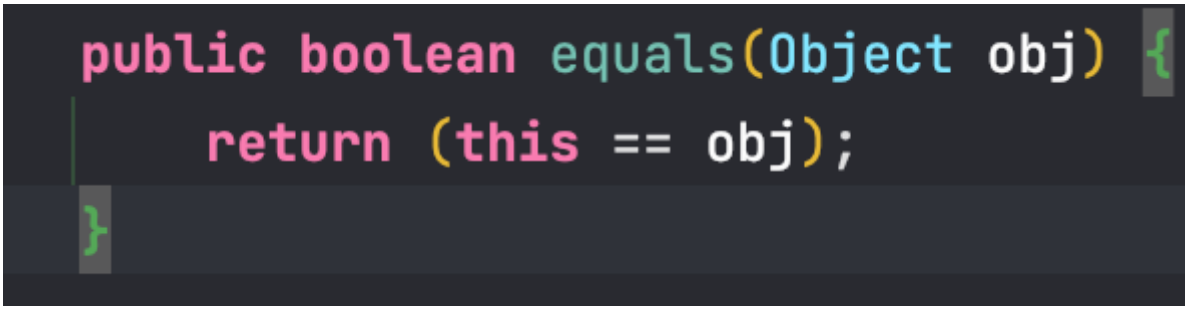
아이템 11을 보면 아래와 같이 언급하고 있다.

`@equals` 를 재정의한 모든 클래스에서 `hashCode`도 재정의해야한다.

> Q: equals로도 논리적 동치비교가 충분히 가능할 것 같은데 왜 hashCode를 사용하는 걸까?

기본적인 equals()는 무엇을 할까

Object 클래스에서 정의한 equals 메서에서는 참조값(즉 주소값)이 같은지만을 확인하고 있다.



하지만 우리는 참조값이 같은지를 보고 싶은 게 아니라

객체의 필드가 모두 같으면 모두 같은 객체로 보고자 합니다.

Point.class, PointWithEquals.class 코드 ㄱㄱㄱ / PointTest 1번 ㄱㄱㄱㄱㄱ

```
public class PointWithEquals {
  int x;
  int y;

  public PointWithEquals(int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public boolean equals(Object o) {
```

```
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    PointWithEquals point = (PointWithEquals) o;
    return x == point.x && y == point.y;
  }
}
```

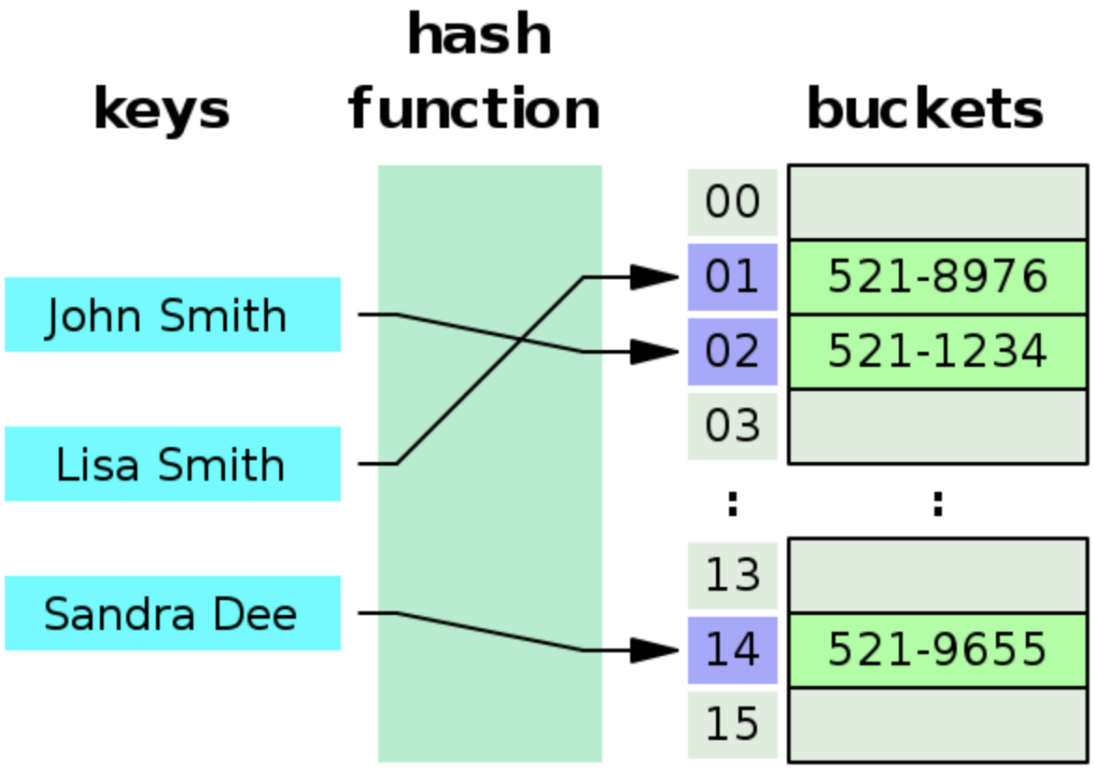
따라서 우리는 이를 재정의하여 참조값(동일성) 뿐만 아니라 값(동등성)까지 부여하였다.

그래서 저는 이 부분에서 equals 메서드만으로도 충분히 논리적 동치비교가 가능할 것 같은데 왜 equals()메서드를 정의할 때 hashCode도 재정의하라는 걸까? 라는 의문이 들었습니다.

hashCode를 사용하는 이유는?

그전에 자료구조 hash에 대한 개념부터 짚고 넘어가자

[Hash 자료구조]



주어진 키에 대해 hash값을 계산하고, 내부적으로 이 값을 사용하여 데이터를 저장한다.

HashTable, HashSet, HashMap 같은 클래스의 이름만 봐도 해시 코드를 사용한다는 것을 짐작할 수 있다.

이처럼 Java에서 해시 코드를 기반으로 데이터를 관리하는 이유는 데이터 검색작업의 시간 복잡도가 O(1)으로 상당히 빠르게 동작하기 때문이다.

[hashCode를 재정의해주라는 의미는?]

equals를 오버라이딩할 때 hashCode도 해줘야 한다는 것은 아래와 같은 것을 의미한다고 생각한다.(p.68에서도 언급)

equals로는 동일한 객체라고 나와서 hashCode를 사용하는 Collection에 같은 키에서 값을 꺼낼 수 있을 것이라고 생각했는데 값을 꺼내지 못하는 경우가 발생한다!!! 왜??? 분명히 Collection에 값이 들어있는데???

PointEquals.class 코드 ㄱㄱㄱㄱ / PointTest 2번 코드 ㄱㄱㄱㄱ

```
@Test
@DisplayName("2. hashCode를 재정의하지 않으면 같은 객체라도 다른 해시 버킷값을 가진다.")
void notHashCode() {
  PointWithEquals p1 = new PointWithEquals( x: 1,  y: 2);
  PointWithEquals p2 = new PointWithEquals( x: 1,  y: 2);
  HashSet<PointWithEquals> set = new HashSet<>();

  // 넣는 Object의 hashCode 값을 구해서 해당 hashCode값에 맞는 해시 버킷에 값을 넣는다.
  set.add(p1);
  set.add(p2);

  Assertions.assertEquals( expected: false, set.contains(new PointWithEquals( x: 1,  y: 2)));
  Assertions.assertNotEquals( unexpected: 1, set.size());
}
```

논리적 동치인 인스턴스를 통해서 set에서 값을 꺼내려고 했지만 꺼내지 못한다.

왜냐하면 이전 인스턴스는 논리적 동치인 새로운 인스턴스와 hashCode값이 서로 다르기 때문이다.

따라서 서로 다른 해시 버킷에 가서 객체를 찾으려고하기에 객체를 찾지 못한다.

이렇게 자바에서는 객체의 빠른 탐색을 위해서 hashCode를 기반으로 동작하는 다양한 Collection을 사용하고

이러한 Collection을 이용해서 우리가 생각한 것처럼 동작하기 위해서는 equals로 동치성이 입증된 객체들은 hashCode도 동일해야 했습니다~

```java
@Test
@DisplayName("2. hashCode를 재정의하면 동등성이 보장되는 인스턴스는 동일한 hashCode값을 가진다")
void equalsAndHashCode() {
    PointWithEqualsAndHashCode p1 = new PointWithEqualsAndHashCode( x: 1,  y: 2);

    HashSet<PointWithEqualsAndHashCode> set = new HashSet<>();
    set.add(p1);

    Assertions.assertEquals( expected: true, set.contains(new PointWithEqualsAndHashCode( x: 1,  y: 2)));
}
```

[그럼 모든 객체들이 동일한 해시값을 가진다면?]

hashCode값이 모두 동일하면 우리가 예상한 것처럼 hashCode를 기반으로 동작하는 Collection을 사용할 수 있습니다.

바로, 모든 객체가 동일한 42 버킷(여기서 PointWithSameHashCode 객체의 hashCode는 42) 들어가서 LinkedList에서 값을 꺼내 equals를 이용해 동일한 인스턴스를 찾기 때문입니다.

그러면 우리는 왜 모든 객체에 대해 동일한 hashCode를 부여하면 안 되는 걸까요?

PointWithSameHashCode.class 코드 ㄱㄱㄱ/ PointTest 3번 코드 ㄱㄱㄱ

```java
@Test
@DisplayName("3. hashCode값이 모든 객체에 대해 동일하면 hashCode를 사용하는 자료구조에서 해당 객체를 찾을 수 있다.")
public void allSameHashCode() {
    PointWithSameHashCode p1 = new PointWithSameHashCode( x: 1,  y: 2);
    PointWithSameHashCode p2 = new PointWithSameHashCode( x: 2,  y: 1);
    HashSet<PointWithSameHashCode> set = new HashSet<>();

    set.add(p1);
    set.add(p2);

    // 전혀 다른 Object임에도 불구하고 해시코드 값이 동일하다 -> 해시 충돌
    Assertions.assertEquals(p1.hashCode(), p2.hashCode());
    Assertions.assertEquals( expected: true, set.contains(new PointWithSameHashCode( x: 2,  y: 1)));
}
```

p68에서는 모든 객체에 동일한 hashCode값을 주는 것을 권장하지 않는 이유를 다음과 같이 이야기합니다.

> 모든 객체에게 똑같은 값만 내어주므로 모든 객체가 해시테이블의 하나의 버킷에 담겨 마치 연결 리스트처럼 동작한다. 그 결과 평균 수행 시간이 O(1)인 해시테이블이 O(n)으로 느려져서, 객체가 많아지면 도저히 쓸 수 없게 된다.

즉, 탐색이 빠르다는 hash의 장점을 쓰지 못한다. 라는 의미인데요.

아!!! 탐색 시간 느려지겠네~ 라는 생각은 들지만, 진짜 느려져???라는 생각이 들어 한 번 코드를 통해 알아봅시다~

(엄청나게 많은 데이터를 생성해야 하기 때문에 이번주 카테캠에서 언급한 EasyRandom 라이브러리를 이용해서 대량의 테스트 데이터를 만들어주었습니다)

```java
@Test
@DisplayName("hashCode의 값이 모든 객체에 대해 동일하면 hashCode를 사용하는 자료구조에서 해당 객체를 찾는데 시간이 오래걸린다.")
void sameHashCodeHighTime() {  Complexity is 3 Everything is cool!
    long start = System.currentTimeMillis();

    Set<PointWithSameHashCode> set = new HashSet<>();
    EasyRandom easyRandom = new EasyRandom();
    easyRandom.objects(PointWithSameHashCode.class,  streamSize: 1000000000).filter(set::add);
    set.add(new PointWithSameHashCode( x: 1,  y: 2));

    boolean isContain= set.contains(new PointWithSameHashCode( x: 1,  y: 2));
    long end = System.currentTimeMillis();

    System.out.println(end - start);
    Assertions.assertEquals( expected: true, isContain);
}
```

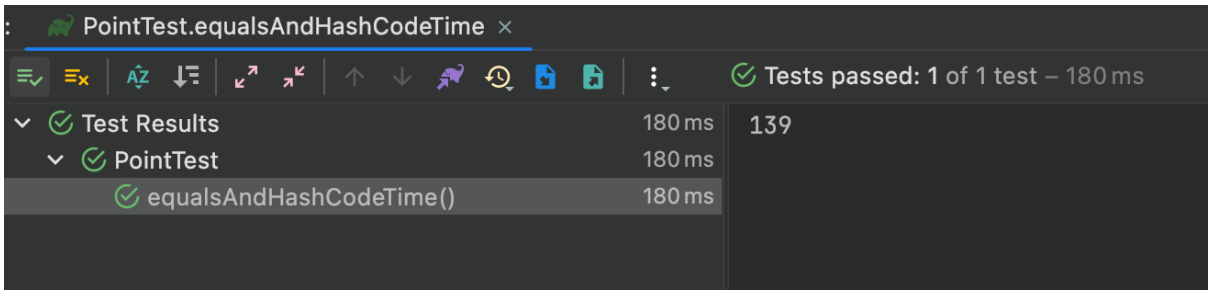
```
PointTest.sameHashCodeHighTime ×
Tests passed: 1 of 1 test – 198 ms
Test Results                198 ms   161
  PointTest                 198 ms
    hashCode의 값이 모든 객체에 대해 동일하면 hasl 198 ms
```

```java
@Test
void equalsAndHashCodeTime(){ Complexity is 3 Everything is cool!
  long start = System.currentTimeMillis();

  Set<PointWithEqualsAndHashCode> set = new HashSet<>();
  EasyRandom easyRandom = new EasyRandom();
  easyRandom.objects(PointWithEqualsAndHashCode.class, streamSize: 1000000000).filter(set::add);
  set.add(new PointWithEqualsAndHashCode( x: 1, y: 2));

  boolean isContain= set.contains(new PointWithEqualsAndHashCode( x: 1, y: 2));
  long end = System.currentTimeMillis();

  System.out.println(end - start);
  Assertions.assertEquals( expected: true, isContain);
}
```



저희가 생각한 것처럼 객체에 따라 다른 hashCode값을 부여한 경우 시간이 더 적게 드는 것을 확인할 수 있었습니다.

그런데!! 생각보다는 차이가 많이 안 난다 라는 생각이 들었습니다.

저는 그래서 이 테스트의 문제점으로

1. 테스트 데이터의 개수가 작다.
2. 한 해시 버킷에 데이터가 많이 담길 경우 HashSet에서 다르게 동작할 수도 있다.

두 가지 케이스가 있을 수 있다는 생각이 들었습니다.

HashMap Performance Improvements in Java 8

[해시맵 내부의 연결 리스트]

> 자바 8 에서 해시 충돌 시 성능 개선을 위해 내부적으로 동일한 버킷에 일정 개수 이상(8개 이상)의 엔트리가 추가되면, LinkedList 대신 binary search를 사용하도록 바뀌었다.

따라서 탐색을 하는 데 걸리는 시간을 줄일 수 있었습니다.

물론, 우리가 이렇게까지 해시 충돌이 많이 일어나는 것을 겪기는 힘들 수 있지만

이러한 자바의 변화에 대해서 학습하고 인지하는 것은 좋겠다고 생각하여 가지고 왔습니다~

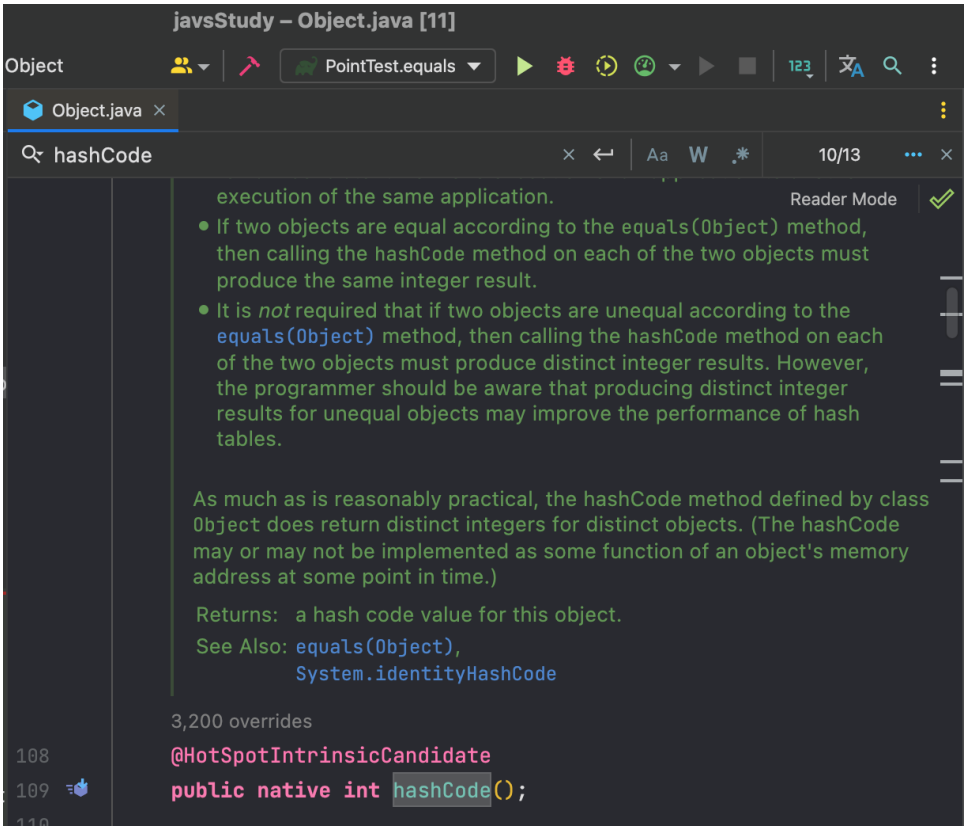
hashCode ≠ memeory address

Java에서 사용되는 해시 코드(Hashcode)는 객체를 식별하기 위한 ID입니다. Java의 모든 객체는 JVM에 의해 고유 번호가 생성되며, 이 고유 번호가 해시 코드입니다.

해시 코드는 32 비트 고유한 정수 값으로 객체와 다른 객체를 구별하기 위해 사용되며, 객체의 내부 주소를 정수로 변환된 값입니다

[hashCode의 구현체를 봐보자]

Object class에서 hashCode메서드를 봅시다!



native method

Object의 hashCode() 메서드는 `native method` 이며 구현은 순수 Java가 아닙니다.
Object 클래스의 소스를 살펴보면 해시코드에 대한 다음 코드를 확인할 수 있습니다.

그럼 hashCode 메서드의 주석을 봐봅시다.

> As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (The hashCode may or may not be implemented as some function of an object's memory address at some point in time.)
>
> 클래스 Object에 의해 정의된 hashCode 메서드는 고유한 개체에 대해 고유한 정수를 반환합니다. (hashCode는 특정 시점에서 객체의 메모리 주소의 일부 기능으로 구현되거나 구현되지 않을 수 있습니다.)

즉, **hashCode는 고유한 개체에 고유한 정수를 반환을 하지, 메모리주소로 무조건 구현되지는 않는다는 말이다.**

hashCode와 메모리주소는 관계가 없다. 라고 말하고 있는 것 같습니다.

그럼 이제 다양한 레퍼런스에서 얻은 결과를 같이 알아볼까요?

Does hashcode number represent the memory address?

> The hashcode for objects is implementation-specific, but I would highly doubt any JVM implementation would use the memory address. Since garbage collection is a central feature of Java, that means the object could be moved and thereby have different memory addresses during its lifetime, even if its contents remain unchanged (and this would violate the hashcode spec).

`객체의 해시코드는 구현에 따라 다르지만, 어떤 JVM 구현도 메모리 주소를 사용할 것 같지는 않습니다.` 가비지 컬렉션은 Java의 핵심 기능이기 때문에 객체의 내용이 변경되지 않더라도 객체가 이동되어 수명 기간 동안 다른 메모리 주소를 가질 수 있습니다(이는 해시코드 사양에 위배됩니다).

위 답변을 보고 hashCode와 메모리 주소를 같다고 볼 수 없는 이유를 명확하게 알아차릴 수 있었습니다.

위에서 언급한 것처럼 가비지 컬렉터에 의해 객체의 메모리 주소는 지속적으로 변합니다.

이와 반대로 hashCode는 p67에서 언급한 것처럼 equals 비교에 사용되는 정보가 변경되지 않았다면 객체의 hashCode 메서드도 같은 값을 반환해야 합니다.

따라서 hashCode를 메모리 주소와 같다고 생각한다면, hashCode 규약에 어긋나게 됩니다.

[stack overflow] difference between hash code and reference or address of an object?

> Hashcode is a number used by JVM for hashing in order to store and retrieve the objects. For example, when we add an object in hashmap, JVM looks for the hashcode implentation to decide where to put the object in memory. When we retrieve an object again hashcode is used to get the location of the object. Note that hashcode is not the actual memory address but its a link for JVM to fetch the object from a specified location with a complexity of O(1).

해시코드는 오브젝트를 저장하고 검색하기 위해 JVM에서 해싱에 사용하는 숫자입니다. 예를 들어 해시맵에 객체를 추가하면 JVM은 해시코드 함축을 찾아 메모리에서 객체를 어디에 넣을지 결정합니다. 객체를 다시 검색할 때 해시코드는 객체의 위치를 가져오는 데 사용됩니다. `해시코드는 실제 메모리 주소가 아니라 JVM이 지정된 위치에서 O(1)의 복잡도를 가진 객체를 가져올 수 있는 링크` 라는 점에 유의하세요.

💡 즉, hashCode는 객체의 메모리 주소가 아니라
객체의 메모리 주소를 어떻게 더 빠르게 찾아갈까? 라는 생각에서 나온 게 hashCode입니다.

그럼 우리가 hashCode값을 정의해주지 않았을 때 hashCode 메서드에서 기본적으로 제공하는 값은 무슨 기준일까?

How does the default hashCode() work?

native 쪽을 살펴보기 위해 C 코드(Object.c)를 살펴보면 아래와 같이 매핑된 함수를 찾을 수 있다.

```
static JNINativeMethod methods[] = {
    {"hashCode",    "()I",                    (void *)&JVM_IHashCode},
    {"wait",        "(J)V",                   (void *)&JVM_MonitorWait},
    {"notify",      "()V",                    (void *)&JVM_MonitorNotify},
    {"notifyAll",   "()V",                    (void *)&JVM_MonitorNotifyAll},
    {"clone",       "()Ljava/lang/Object;",   (void *)&JVM_Clone},
};
```

hashCode 는 JVM_IHashCode 로 매핑이된다.

JVM_IHashCode 는 jvm.h/cpp 에 정의 및 구현이 되어 있다.

```
JVM_ENTRY(jint, JVM_IHashCode(JNIEnv* env, jobject handle))
  JVMWrapper("JVM_IHashCode");
  return handle == NULL ? 0 : ObjectSynchronizer::FastHashCode (THREAD, JNIHandles::resolve_non_nu
ll(handle)) ;
JVM_END
```

ObjectSynchronizer 클래스의 FastHashCode 메서드를 호출하는 것을 알 수 있다.

FastHashCode 는 내부적으로 get_next_hash 메서드를 호출한다.

```
  monitor = ObjectSynchronizer::inflate(Self, obj, inflate_cause_hash_code);
  mark = monitor->header();
  hash = mark->hash();
  if (hash == 0) {
    hash = get_next_hash(Self, obj);
    temp = mark->copy_set_hash(hash);
    test = Atomic::cmpxchg(temp, monitor->header_addr(), mark);
    if (test != mark) {
      hash = test->hash();
    }
  }
  return hash;
}
```

```
intptr_t ObjectSynchronizer::FastHashCode(Thread * Self, oop obj) {
  if (UseBiasedLocking) {
    if (obj->mark()->has_bias_pattern()) {
      Handle hobj(Self, obj);
      BiasedLocking::revoke_and_rebias(hobj, false, JavaThread::current());
      obj = hobj();
    }
  }

  ObjectMonitor* monitor = NULL;
  markOop temp, test;
  intptr_t hash;
  markOop mark = ReadStableMark(obj);

  if (mark->is_neutral()) {
    hash = mark->hash();
    if (hash) {
      return hash;
    }
    hash = get_next_hash(Self, obj);
    temp = mark->copy_set_hash(hash);
    test = obj->cas_set_mark(temp, mark);
    if (test == mark) {
      return hash;
    }
  } else if (mark->has_monitor()) {
    monitor = mark->monitor();
    temp = monitor->header();
    hash = temp->hash();
    if (hash) {
      return hash;
    }
  } else if (Self->is_lock_owned((address)mark->locker())) {
    temp = mark->displaced_mark_helper();
    hash = temp->hash();
    if (hash) {
      return hash;
    }
  }
```

```
static inline intptr_t get_next_hash(Thread * Self, oop obj) {
  intptr_t value = 0;
  if (hashCode == 0) {
    // This form uses global Park-Miller RNG.
    // On MP system we'll have lots of RW access to a global, so the
    // mechanism induces lots of coherency traffic.
    value = os::random();
  } else if (hashCode == 1) {
    // This variation has the property of being stable (idempotent)
    // between STW operations.  This can be useful in some of the 1-0
    // synchronization schemes.
    intptr_t addrBits = cast_from_oop<intptr_t>(obj) >> 3;
    value = addrBits ^ (addrBits >> 5) ^ GVars.stwRandom;
  } else if (hashCode == 2) {
    value = 1;            // for sensitivity testing
  } else if (hashCode == 3) {
    value = ++GVars.hcSequence;
  } else if (hashCode == 4) {
    value = cast_from_oop<intptr_t>(obj);
  } else {
    // Marsaglia's xor-shift scheme with thread-specific state
    // This is probably the best overall implementation -- we'll
    // likely make this the default in future releases.
    unsigned t = Self->_hashStateX;
    t ^= (t << 11);
    Self->_hashStateX = Self->_hashStateY;
    Self->_hashStateY = Self->_hashStateZ;
    Self->_hashStateZ = Self->_hashStateW;
    unsigned v = Self->_hashStateW;
    v = (v ^ (v >> 19)) ^ (t ^ (t >> 8));
    Self->_hashStateW = v;
    value = v;
  }

  value &= markOopDesc::hash_mask;
  if (value == 0) value = 0xBAD;
  TEVENT(hashCode: GENERATE);
  return value;
}
```

get_next_hash라는 메서드의 알고리즘은 주어진 전역변수 hashCode가 설정된 값에 따라 여러가지 알고리즘으로 분기한다.

0. 운영체제에 구현된 random(Park-Miller RNG)을 사용
1. 객체의 메모리 주소 int로 캐스팅 한 값을 우로 3 쉬프트 한 값을 사용
2. 고정된 값 1을 반환 (테스팅 목적)
3. 연속된 시퀀스를 반환
4. 객체의 메모리 주소를 그냥 int로 캐스팅함
5. XOR 쉬프트를 통한 스레드 상태를 기반으로 생성

그러면 hashCode는 어떤 값에 의해 선택될까?

globals.hpp에 따르면 OpenJDK 8의 경우 5번이 선정된다.

```
#define RUNTIME_FLAGS(develop, develop_pd, product, product_pd, diagnostic, experimental, notprodu
ct, manageable, product_rw, lp64_product) \
// ..
  develop(bool, InlineObjectHash, true,                                     \
          "Inline Object::hashCode() native that is known to be part "      \
          "of base library DLL")                                            \
  product(intx, hashCode, 5,                                                \
          "(Unstable) select hashCode generation algorithm")                \
```

그런데 JDK 6, 7에서는 0번의 전략을 선택했다. (src/share/vm/runtime/globals.hpp)n

```
develop(bool, InlineObjectHash, true,                                    \
        "inline Object::hashCode() native that is known to be part "     \
        "of base library DLL")                                           \
product(intx, hashCode, 0,                                               \
       "(Unstable) select hashCode generation algorithm" )               \
```