Java Program Performance Tuning

김명호 기술이사
㈜비트웹
toodull@hitel.net

목 차

자바와 성능

- 프로그램의 성능이 나쁜 이유
  - 성능 조정을 수행하지 않았기 때문
  - 비효율적인 알고리즘이나 데이터 구조 사용
- 자바 프로그램의 성능이 나쁜 기타 이유
  - 플랫폼 독립성을 위한 소프트웨어 레이어
  - 재사용/유연성/안전성을 위한 여러 동적 특성:
  - 참조 의미론, 힙 할당, 가비지 수집, 동적 바인딩, 형 변환과 첨자에 대한 철저한 검사...

자바와 성능 (계속)

- 자바는 무조건 좋은 언어라는 자바 원리주의자가 되지 말라
  - 누구를 위한 좋은 언어인가?
  - 프로그램 효용성은 사용자가 결정하는 것이다
  - 자바 프로그램이기만 하면 아무리 비효율적이라도 사용자가 만족할까?
- 자바의 핵심 API는 가장 효율적으로 구현되었을 것으로 맹목적으로 믿지 말라
  - 이들이 성능을 저해하는 주된 원인이 될 수 있다

성능 조정의 일반 원칙

- 성능 조정의 목표
  - 사용자가 받아들일 수 있는 수준의 성능
  - 사용자 관점에서의 성능을 추구할 것 (성능=반응성)
  - 성능을 좌우하는 부분을 집중 조정할 것 (8:2)
- 임의적 작업이 아니라 과학적 과정 적용
  - 주된 병목(bottleneck)을 판정한다
  - 손쉽게 해결할 수 있는 병목을 선정, 집중 개선한다
  - 이러한 과정을 사용자가 만족할 때까지 반복한다

성능 조정 예제
처음 n개의 소수 (Primes0)

```
static List primes(int n)
{
    List primes = new ArrayList(n);
outer:
    for (int candidate = 2; n > 0; candidate++) {
        Iterator iter = primes.iterator();
        while (iter.hasNext()) {
            int aprime = ((Integer)iter.next()).intValue();
            if (candidate%aprime == 0)
                continue outer;
        }
        primes.add(new Integer(candidate)); n--;
    }
    return primes;
}
```

성능 측정

- 짐작에 의존하지 말고 반드시 측정하라
- 측정 대상
  - 성능에 영향을 미칠 수 있는 모든 요인
  - 실행시간, 기억장소, 데이터 이동, 프로그램 규모
- 성능 조정 후에는 반드시 효과를 측정하라

실행시간 측정

- **System.currentTimeMillis() 사용**

```
long startTime = System.currentTimeMillis();
//…실행 시간을 측정할 코드...
long duration = System.currentTimeMillis() - startTime;
```

- java Primes0 *count*
  - count ? 100: 0 ms
  - count = 1,000: 170 ms
  - count = 10,000: 15,320 ms
  - count = 100,000: 5,850,320 ms

실행시간 측정시 유의사항

- 프로파일링, 디버깅 모드를 해제할 것
  - 이들은 실행시간에 부정적인 영향이 있음
- 개발 환경이 아니라 디플로이 환경에서 측정
  - 개발 환경과 실제 환경은 성능 차이가 클 수 있음
- 여러 번 측정한 후 평균할 것
  - 초기화 비용, 캐쉬 효과, 시스템 로드 등의 영향이 있을 수 있음

기억장소 측정

- 가비지 수집 발생 추적
  - java 명령에 **-verbosegc …** 옵션을 사용하여 확인

- java -verbosegc Primes0 10000
- [GC 511K->158K(1984K), 0.0061533 secs]
- [GC 670K->190K(1984K), 0.0030096 secs]
- [GC 702K->220K(1984K), 0.0023509 secs]
- [GC 732K->249K(1984K), 0.0040178 secs]
- [GC 761K->278K(1984K), 0.0033440 secs]
- …

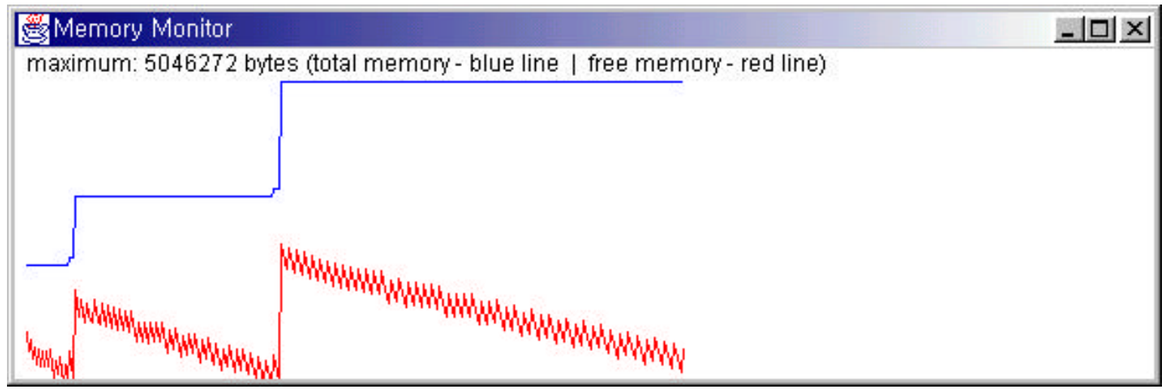
기억장소 측정 (계속)

- 객체 생성 조회
  - Object 클래스를 변경하여 객체 생성을 추적할 수 있는 생성자 추가

- java -Xbootclasspath/p:object.jar Primes0 10000
  - java.lang.Integer 10,000
  - java.util.ArrayList 1
  - java.util.AbstractList$Itr 104,728

기억장소 측정 (계속)

- 기억장소 전체 사용량
  - **Runtime** 클래스의 **totalMemory()**, **freeMemory()** 메소드 사용



- 그래프에서 리플은 가비지 수집이 발생한 증거임

병목 판정

- java **–Xrunhprof:cpu=samples** …
  - **프로파일 요약 테이블과 트레이스 정보 제공**

- **프로파일 요약 테이블**
- CPU SAMPLES BEGIN (total = 659) Sun Jan 20 20:23:05 2002

```
rank   self  accum count trace method
   1 49.01% 49.01%   323 13    Primes0.primes
   2 21.55% 70.56%   142 12    java.util.AbstractList$Itr.next
   3  7.74% 78.30%    51 11    java.util.ArrayList.get
   4  7.44% 85.74%    49 14    Primes0.primes
   5  7.28% 93.02%    48  9    Primes0.primes
   6  4.55% 97.57%    30 15    java.util.AbstractList$Itr.hasNext
   7  1.06% 98.63%     7 10    java.util.AbstractList.iterator
   8  0.15% 98.79%     1 16    Primes0.primes
   …
```

- **상위 2개 요소가 실행 시간의 70%를 소요함**

병목 판정 (트레이스)

- TRACE 13
- `//while (iter.hasNext()) ...`
- Primes0.primes(Primes0.java:22)
- Primes0.main(Primes0.java:9)

- TRACE 12
- `//int aprime = ((Integer)iter.next()).intValue();`
- java.util.AbstractList$Itr.next(<Unknown>:Unknown line)
- Primes0.primes(Primes0.java:23)
- Primes0.main(Primes0.java:9)

병목 정보의 활용

- 병목 개선
  - 병목 코드의 내부가 복잡하고 비효율적인 경우
  - 병목 코드에 대한 수정 권한이 있는 경우
- 병목 회피
  - 병목 코드가 매우 빈번히 호출되는 경우
  - 병목 원시 코드를 직접 수정할 권한이 없는 경우

병목 정보의 활용

- Primes0의 주요 병목
  - 약수 발견을 위한 while 문 반복 회수 감소 필요
- 이론적인 고찰
  - 어떤 수의 소수 약수는 그 수의 제곱근 이하이다.
- Primes0에 대한 적용
  - 약수 판별 논리 부분에 다음과 같은 조건 추가
    - `if (aprime*aprime > candidate) break;`
    - `[Not if (aprime > (int)Math.sqrt(candidate)) break;]`
  - 알고리즘 변경을 통한 병목 직접 개선

처음 n개의 소수 (Primes1)

```
static List primes(int n)
{
    List primes = new ArrayList(n);
outer:
    for (int candidate = 2; n > 0; candidate++) {
        Iterator iter = primes.iterator();
        while (iter.hasNext()) {
            int aprime = ((Integer)iter.next()).intValue();
            //Applying better algorithm
            if (aprime*aprime > candidate) break;
            if (candidate % k == 0)
                continue outer;
        }
        primes.add(new Integer(candidate)); n--;
    }
    return primes;
}
```

Primes1의 성능 특성

- 실행시간 (java Primes1 *count*)
  - count ? 100: 0 ms (vs. 0 ms)
  - count = 1,000: 50 ms (vs. 170 ms)
  - count = 10,000: 220 ms (vs. 15,320 ms)
  - count = 100,000: 4,450 ms (vs. 5,850,320 ms)
- 기억장소
  - Primes0의 경우와 동일함
  - Integer, Iterator(Itr) 객체 생성 최적화 필요

HotSpot VM

- 세대별 가비지 수집 알고리즘 사용
  - 수명이 매우 짧은 객체들이 많은 경우 효율적
- 핸들(포인터의 포인터) 대신 직접 포인터 사용
- 객체 생성과 초기화, 스레드 동기화 개선
- 적응적 최적화 (adaptive optimization)
  - 실행 시에 수집된 프로파일 데이터를 기반으로
  - 병목 주변 메소드를 JIT 컴파일러로 컴파일함
  - 순수 JIT 에 비해 적극적으로 최적화를 시도

javac와 최적화

- javac가 기본적으로 수행하는 최적화
  - String 접속 연산의 해석
  - final로 선언된 변수(=상수)의 폴딩과 전파
  - 도달 불가능한 코드의 제거:
  - 상수 전파와 연관될 때 “조건부 컴파일”의 효과
- 기타 최적화는 거의 시도하지 않음
  - HotSpot VM에 최적화 기능이 집중되어 있음
- -O 옵션도 실행 효율에 영향이 거의 없음

주의, 주의, 주의! ☞

- 이후의 모든 기술은 병목에만 적용하라
  - 성능에 문제가 되지 않는 부분은 그대로 두는 것이 최선의 최적화이다
- 최적화 기술의 효과에 너무 의존하지 말라
  - 알고리즘과 데이터 구조의 선택이 보다 중요하다
- 기술 적용 후 그 효과를 반드시 측정하라
  - 최적화 기술이 항상 성능을 개선하는 것은 아니다
- 기존 버전과 기능 동일성을 항상 확인하라
  - 신속히 오답을 구하는 프로그램은 가치가 없다!

일반적인 최적화

- **연산 강도 축소** (strength reduction)
  - **예: 0과의 비교는 다른 어떤 비교보다 효율적이다**
    - `for (int i = 0; i < size; i++) …`
      - `for (int i = size-1; i >= 0; i--) …`
- **루프 불변식 이동** (code motion)
  - **예: l이 List이고 l.size()가 고정된 값인 경우**
    - `for (int i = 0; i < l.size(); i++) …`
      - `for (int i = 0, lsize = l.size(); i < lsize; i++) …`

일반적인 최적화 (계속)

- 상수 폴딩과 전파
  - `int max_a = 100;`
  - `int max_b = 200;`
  - `for (int i = arr.length-1; i >= 0; i--)`
    - `arr[i] = max_a + max_b;`

```
final int max_a = 100;
final int max_b = 200;
```

- final 선언의 결과 `arr[i] = max_a + max_b;` 문장이 `arr[i] = 300;` 과 같이 컴파일됨

중복 계산 회피

- 중복 초기화 회피
  - class Poor {
    - int field = 0;
    - public Poor() { field = 0; }
  - } //field is initialized 3 times!

```
data = new int[items];
for (int i = 0; i < items; i++)
    data[i] = 0; //redundant initialization
```

중복 계산 회피 (계속)

중복 액세스 회피

```
MyData data[];
…
data[m+n].doit(a+1);
data[m+n].doit(a+2);
```

```
MyData dmn = data[m+n];
dmn.doit(a+1);
dmn.doit(a+2);
```

```
data[m+n].doit(a+1)
         .doit(a+2);
```

중복 계산 회피 (계속)

- 중복 검사 회피
  - ArrayList.get(int index)
    - `rangeCheck(index);`
    - `return elementData[index];  //redundant check`
  - ArrayList.rangeCheck(int index)
    - `if (index >= size || index < 0) //check once`
    - `  throw new IndexOutOfBoundsException(…);`
    - …

객체 생성

- 객체 생성은 매우 고비용의 연산이다
  - 무분별한 객체 생성은 병목이 되기 쉽다
- 생성 비용은 상속 깊이와 필드 개수에 비례
  - 설계상의 이유로 인하여 이들을 조절하기 어렵다
- 불필요한 객체 생성을 회피하라

객체 생성 (계속)

- 임시 객체의 생성을 피하라
  - 반복문 내에서 임시 객체의 생성을 피하라
  - 객체 생성을 회피할 수 있는 API를 사용하라
    - `new Integer("100").intValue()`
      - `Integer.parseInt("100") //클래스 메소드 사용`
    - 가능한 경우 기존 객체를 재사용하라
- 기본형 데이터를 위한 컬렉션 사용을 자제하라
  - 래퍼 객체의 생성/조작 부담과 컬렉션의 지나친 일반성 때문에 병목이 되기 쉽다
  - 객체보다는 기본형과 배열을 주로 사용하라

String과 StringBuffer

- 스트링의 해석 시기에 주의하라!
  - 컴파일 시와 실행 시
- 컴파일 시에 해석되는 스트링
  - String str = “Hello ” + “world!”;
    - String str = “Hello world!”; //컴파일러가 해석함
  - 컴파일 시에 해석될 수 있는 스트링은 그대로 두는 것이 최상의 최적화이다!
  - 컴파일러에 따라 해석하는 정도의 차이가 있다

String과 StringBuffer (계속)

- 실행 시에 해석되는 스트링
  - String str = a + b; //컴파일 시 해석이 불가능하면
  - String str =new StringBuffer().
  - append(a).append(b).toString(); 로 컴파일 됨
- 연속적인 스트링 접속 연산을 피하라

```
String str = “”;
for (…)
    str += …;
```

```
StringBuffer sbuf =
    new StringBuffer(“”);
for (…)
    sbuf.append(…);
String str = sbuf.toString();
```

  - 임시 객체 생성 회피의 구체적 사례임
- StringBuffer를 적절한 크기로 초기화하라

String과 StringBuffer 비교

- 0;1;2…;n을 위한 스트링 생성 실험
- 실행 시간 비교 (ms)

n	<=100	1,000	10,000	100,000
String	0	280	26,530	6,833,940
StringBuffer	0	50	110	1,150

- 객체 생성 비교
  - String: 3*n +? vs. StringBuffer: n +?

반복과 순환

- 반복 관련 여러 표준 최적화를 적용하라
  - 불변식 이동, 루프 펼치기, …
- 매 반복마다 수행되는 기능을 최소화하라
  - 고비용 연산을 보다 저비용의 연산으로 대체하라
- 순환이 반복보다 항상 비효율적이지는 않다
  - 스택을 사용한 순환 제거는 부담이 더 클 수 있다
  - 캐쉬 사용으로 효율을 크게 개선할 수 있다

예외

- 예외 객체 생성은 매우 비효율적 연산이다
  - 스택 트레이스 설정을 위한 fillInStackTrace() 때문
- 스택 트레이스가 필요하지 않으면 예외 객체를 재사용하라

```
if (…)                                  Exception EXCEPTION =
    throw new Exception(…);                 new Exception(…);
if (…)                                  if (…) throw EXCEPTION;
    throw new Exception(…);             if (…) throw EXCEPTION;
```

- 반복문 이탈에 예외를 사용해도 별 이득이 없다

캐스트

- 다운 캐스트는 실행 시 부담이 있다
- 임시 변수를 사용하여 중복 캐스트를 피하라

```
((HasXYZ)obj).x + ((HasXYZ)obj).y + ((HasXYZ)obj).z
```

➡

```
HasXYZ xyz = (HasXYZ)obj;
xyz.x + xyz.y + xyz.z
```

- 포괄성으로 인한 과도한 캐스트를 피하라
- 포괄적 클래스 대신 구체적 클래스를 사용하라
- 변수의 형을 보다 정확하게 선언하라

변수

- 유효범위에 따라 액세스 비용이 다르다
  - 지역변수가 인스턴스/클래스변수 보다 효율적
  - 배열 원소는 null 여부, 첨자 범위 검사가 요구되는 고비용의 연산이다
- 빈번히 참조되는 고비용의 변수는 지역변수를 사용하여 비용을 감소시켜라

```
for (int i = 0; i < REPEAT; i++) counts[0] += i;
```

⇒

```
int acount = counts[0];
for (int i = 0; i < REPEAT; i++) acount += i;
counts[0] = acount;
```

컬렉션

- 예측되는 최종 크기로 사전 할당하라
  - ArrayList/Vector/Stack, Hashtable, StringBuffer 등은 포화되면 고비용의 재할당/복사/재배치 발생
- 대안이 될 수 있는 데이터 구조로 대체하라
  - 특화된 컬렉션과 배열을 사용하여 캐스트, 메소드 호출, 동기화를 피하라
  - 원소들을 빈번히 참조하는 경우 배열로 변환하여 배열 원소들을 참조하라 (Collection.toArray())
- HashMap과 ArrayList에 동기화 래퍼를 사용하면 Hashtable과 Vector에 비해 성능이 나쁘다

입출력

- 버퍼 기능을 제공하는 스트림을 사용하라
  - 디바이스 스트림은 버퍼 기능을 제공하지 않는다!

- 입력 데이터를 캐쉬하거나 재활용하라

- 불필요하거나 중복 실행되는 입출력을 제거하라
  - 트랜잭션 로그, 에러 메시지, ...

스레드

- 스레드는 자원을 대단히 많이 사용한다
- 불필요하거나 의도하지 않은 동기화를 피하라
  - 데이터를 분산시켜 배타적 액세스 필요성을 줄여라
  - 단일 스레드 프로그램은 Vector/Hashtable 대신 ArrayList/HashMap 을 사용하라
- 저성능의 기능을 별도 스레드에 배정하라
  - 입출력, 네트웍 액세스, 사용자 인터페이스, ...

서버 스레드

- 스레드 생성 비용을 감소 시켜라
  - 수명이 매우 짧은 스레드는 재사용하라
- 과도한 개수의 스레드를 유지하지 말라
  - 스레드 풀 관리자를 사용하여 동시 실행 스레드의 수를 제한하라
- 과도한 개수의 스레드를 일시에 생성하지 말라
  - 서비스 큐 기반의 로드 밸런싱을 추구하라

분산 프로그램

- 전달되는 메시지의 수를 감소시켜라
  - 변경되지 않는 데이터는 캐쉬하라
  - 실제 데이터를 요청하기 위한 “스텁”을 사용하라
  - 메시지를 배치 처리하거나 압축하라
- 데이터 이동을 최적화하라
  - 메소드와 가장 가까운 위치에 데이터를 배치하라 (program partitioning)

프로그램 규모 최소화

- 메소드 바이트코드 카운트를 최소화 하라
  - 0 혹은 null을 대상으로 하는 적재 및 비교 연산
  - 인스턴스 변수보다 지역변수를 참조하는 연산
- javac에 –g:none 옵션을 사용하라
- 멤버 명칭, 상수 풀 크기를 축소하기 위하여 shrinker를 사용하라
- obfuscator도 프로그램 축소 효과가 있다

처음 n개의 소수 (Primes2)

```
//Using specialized collection
static intList primes(int n)
{
    intList primes = new intList(n);
outer:
    for (int candidate = 2; n > 0; candidate++) {
        final int size = primes.size();
        //Indexing r.t. Enumeration
        for (int i = 0; i < size; i++) {
            int aprime = primes.get(i);//No cast
            if (aprime * aprime > candidate) break;
            if (candidate % k == 0) continue outer;
        }
        primes.add(candidate);//No object creation
        n--;
    }
    return primes;
}
```

처음 n개의 소수 (Primes2 계속)

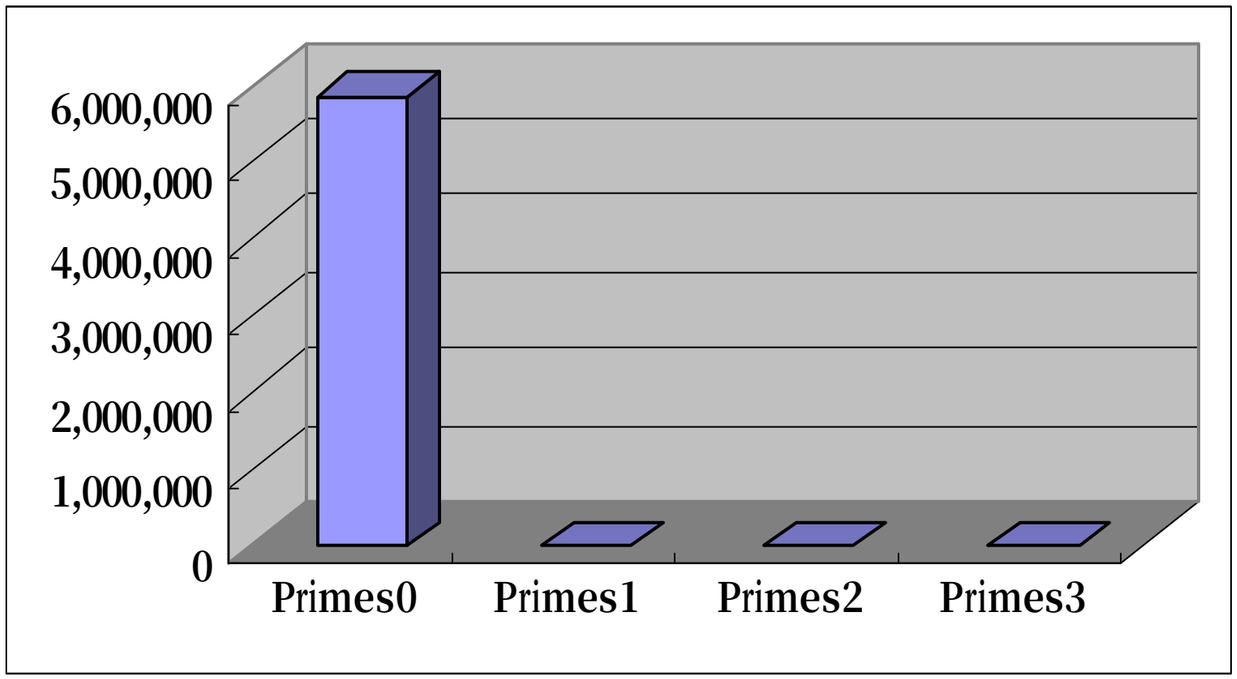
```
class intList
{
    private int[] data;
    private int size, capacity;

    public intList(int capacity) {
        data = new int[capacity]; this.capacity = capacity;
    }
    public int get(int index) {
        //No redundant checks, no cast
        return data[index];
    }
    public int size() { return size; }
    public void add(int  value) { …data[size++] = value;… }
}
```

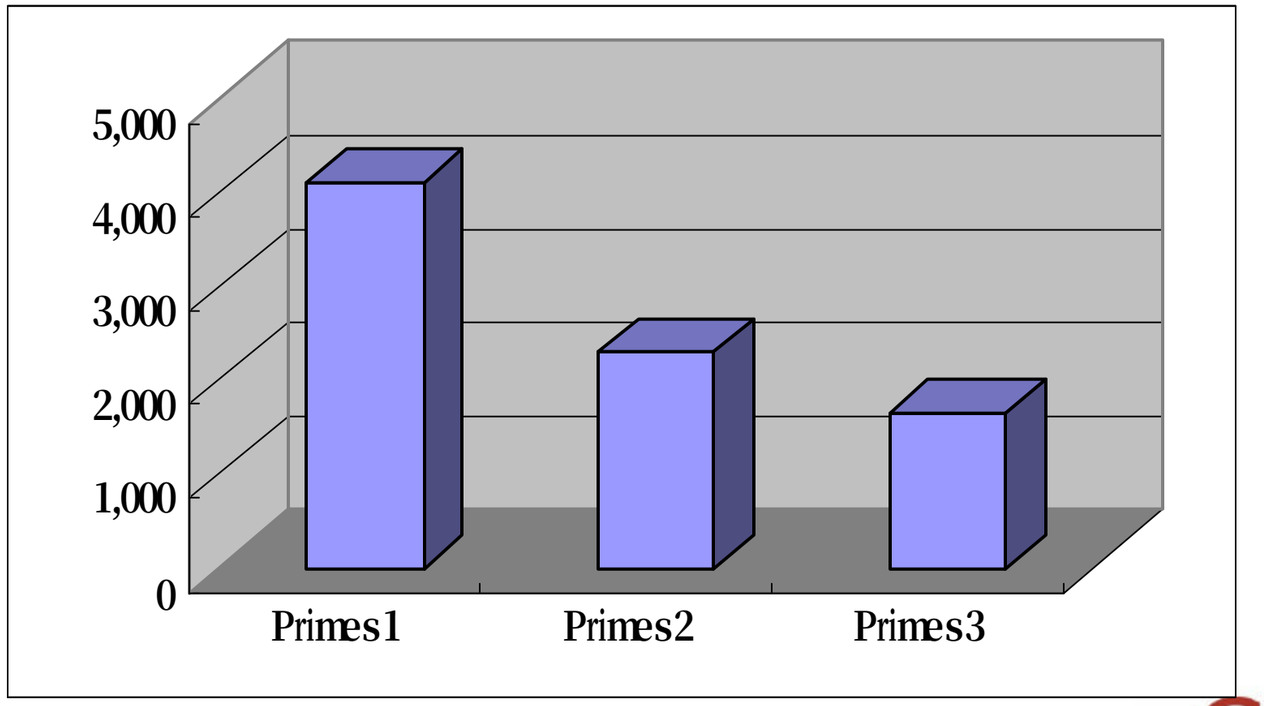
처음 n개의 소수 (Primes3)

```
//Using array instead of collection
static int[] primes(int n)
{
    int[] primes = new int[n];
    int spot = 0;
outer:
    for (int candidate = 2; n > 0; candidate++) {
        for (int i = 0; i < spot; i++) {
            int k = primes[i]; //No method calls, no indirection
            if (k * k > candidate) break;
            if (candidate % k == 0) continue outer;
        }
        primes[spot++] = candidate; n--;
    }
    return primes;
}
```

Primes0-3의 성능 비교
소수 100,000개 생성 (ms)



Primes1-3의 성능 비교
소수 100,000개 생성 (ms)

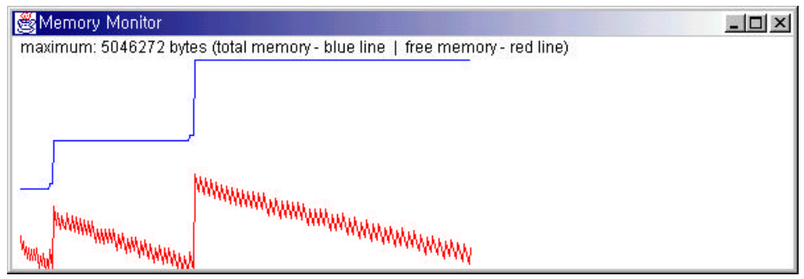


Primes0-3의 객체생성

- Primes0, Primes1
  - java.lang.Integer 100,000
  - java.util.ArrayList 1
  - java.util.AbstractList$Itr 1,299,708
- Primes2
  - intList 1
- Primes3
  - None! --Hurrah!

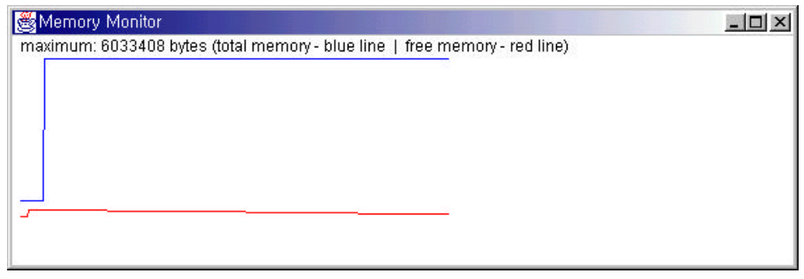
Primes0-3의 전체 기억장소 사용

Primes0, Primes1



빈번한 객체 생성과
가비지 수집 발생

Primes2, Primes3



기억장소 사용이
매우 안정화 되었음

성능 조정의 경험

- 병목 주변의 성능을 수백% 개선하기는 쉽지만 수십%를 더 개선하기는 매우 어렵다
- 알고리즘이 나쁘면 성능이 극도로 나쁘다
  - 성능의 모든 책임을 자바 언어에 미루지 말라
- 핵심 API를 프로그램의 특성에 따라 특화하면 효율을 크게 개선할 수 있다
- 효율 개선을 위하여 프로그램을 변경할수록 유연성과 이해도가 저하된다

결 론

- 자바는 생산성/이식성/안전성 등을 제공하는 매우 우수한 언어이다
  - 그러나 그 대가로 효율성을 희생하는 경우가 있다
  - 성능을 조정하면 사용자가 만족하는 성능을 보일 가능성도 매우 높다
- 성능 공학 (performance engineering)
  - 성능 조정은 임의적 작업이 아닌 체계적 과정이다
  - 병목을 집중 개선하면 큰 효과를 얻을 수 있다