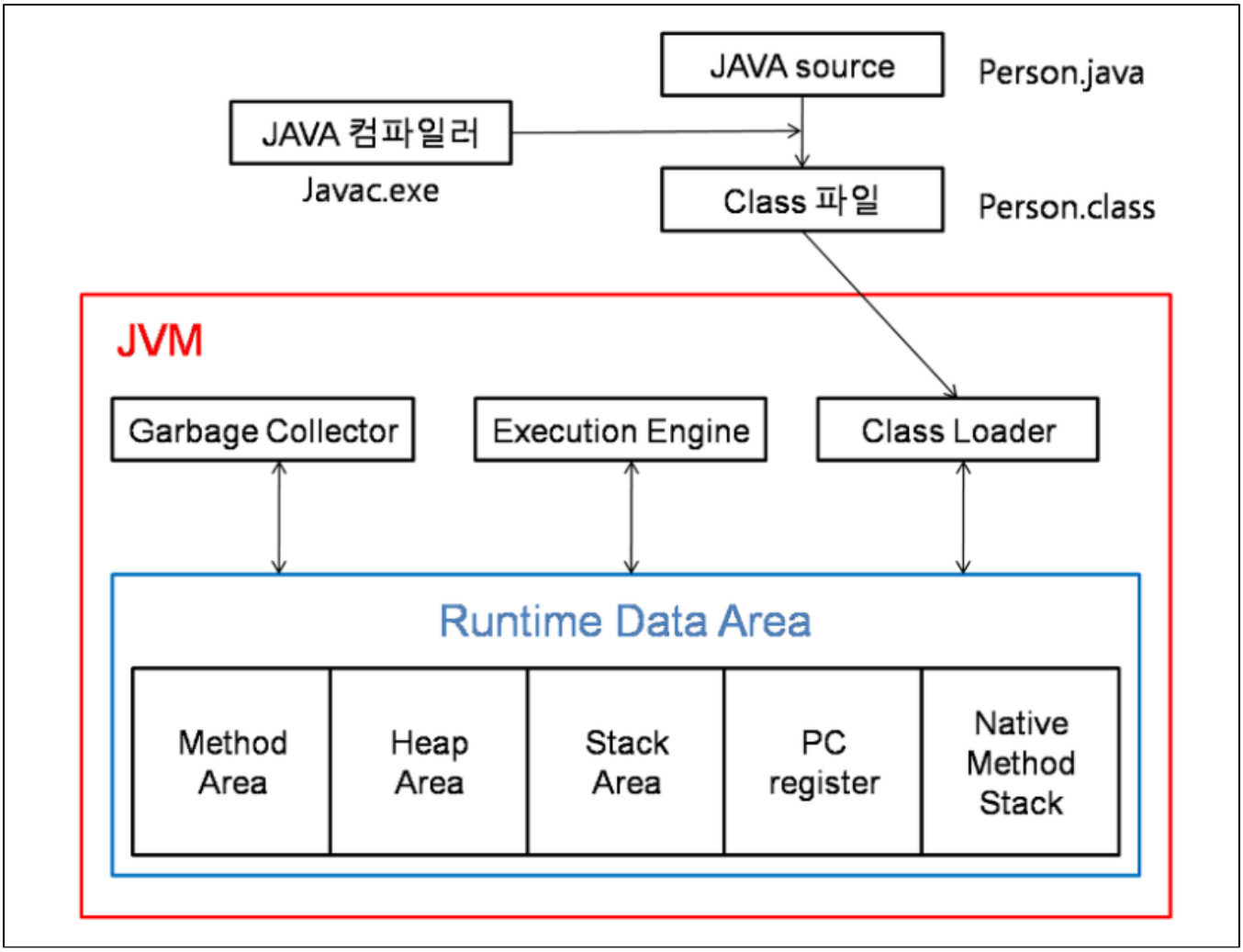
Day 03~04 (2019-01-14)

1. JVM

(1) JMV 구조

Java application 실행 과정

→ Java source를 Java Compiler(javac)가 컴파일
→ 컴파일 산출물인 클래스 파일(byte code)을 Class Loader가 메모리에 로드
→ main 메소드를 찾아 실행 (스택에 메모리 할당, 객체 생성, 객체 소멸, GC ...)



- **Class Loader**
  - 개발자가 작성한 Java 파일을 컴파일러가 컴파일하여 생성한 바이트코드가 존재하는 Class file을 엮어 Run-time Data Area로 적재
  - 클래스 로드의 작업은 Run-time에 동적으로 진행된다

- **Execution Engine**
  - Class Loader에 의해 메모리에 적재된 바이트코드를 기계어로 번역하여 명령어 단위로 실행한다.
    - Interpreter: 바이트코드 명령어를 하나씩 읽어서 해석하고 실행한다. 하나하나의 해석은 빠르지만 실행 결과는 느리다.
    - JIT Compiler: 인터프리터 방식으로 실행하다가 적절한 시점에 바이트코드 전체를 미리 컴파일하여 네이티브 코드로 변경,
      이후에는 해당 메소드를 더이상 인터프리팅 하지 않고 네이티브 코드를 직접 실행한다.
  - 인터프리터와 JIT Compiler 방식은 Trade-off 관계
    → JVM은 내부적으로 해당 메소드가 얼마나 자주 호출되는지 체크하여 컴파일을 수행 함

- **Garbage Collector**
  - 메모리 영역에 생성 된 객체들 중 참조되지 않는 객체들을 탐색하여 제거하는 역할 수행
  - GC가 수행되는 시점은 정확히 예측할 수 없다
  - GC가 수행되는 동안 GC를 수행하는 Thread 외 모든 Thread는 일시 정지 된다

- **Run-time Data Area**
  - JMV이 OS로부터 할당 받은 메모리 영역으로 자바 애플리케이션을 실행할 때 사용되는 데이터를 적재하는 영역

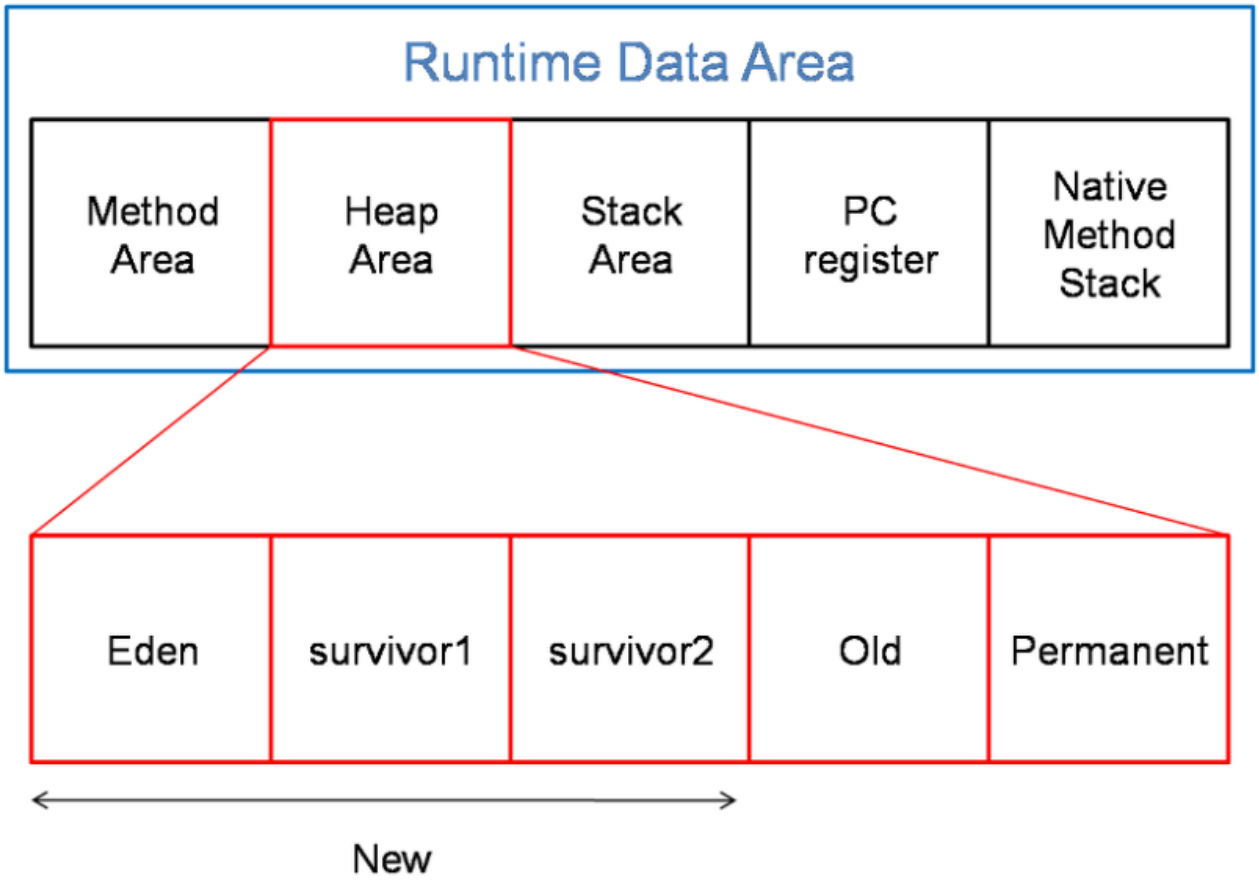
※ Java 최적화?

```
for(int i=0; i< length; i++){
   User u = getUser(i);
}


User u = null;
for(int i=0; i< length; i++){
   u = getUser(i);
}
```

```
for(int i=0; i< length; i++){
   User u = getUser(i);
   callOtherMethod(u);
}


for(int i=0; i< length; i++){
   callOtherMethod(getUser(i));
}
```

(2) Run-time Data Area

자바 런타임 메모리(Runtime Data area)구조



- **Method (Static) Area**
  - Class Loader가 읽어들인 클래스, 인터페이스, 접근제어자, 메소드의 이름, 파라미터, 리턴타입 등의 정보, Constant Pool, static 변수 등이 적재 되는 영역
  - 모든 Thread에서 공유
  - 프로그램 종료시까지 메모리영역이 사용 된다(또는 명시적으로 null 선언 시)

- **Heap Area**
  - new 키워드를 통해 생성된 객체와 배열이 생성되는 영역
  - Method area에 로드 된 클래스에 해당하는 인스턴스만 생성이 가능
  - 모든 Thread에서 공유
  - 객체가 더이상 사용되지 않거나 명시적으로 null 선언시까지 메모리 영역이 사용된다.

- **Stack**
  - 메소드 호출 시 Frame을 추가(Push)하고 메소드가 종료되면 해당 Frame을 제거(Pop)하는 동작을 수행
  - 메소드 정보, 지역변수, 파라미터, 연산 중 발생하는 임시 데이터 저장
  - 기본 타입 변수는 스택 영역에 직접 값을 가지며 참조 타입은 Heap area 또는 Method area이 객체 주소를 가진다

- Thread 마다 하나씩 존재

- **PC Register**
  - 현재 수행중인 JVM 명령 주소를 갖는다
  - 프로그램 실행은 CPU에서 명령어를 수행
  - 연산 결과값을 메모리에 전달하기 전 저장하는 CPU 내의 기억장치

- **Native Method Stack Area**
  - 자바 외 언어로 작성된 네이티브 코드를 위한 Stack
  - JNI(Java Native Interface)를 통해 호출되는 C/C++ 등의 코드를 수행하기 위한 스택
  - Native Method의 파라미터, 지역변수 등을 바이트코드로 저장한다

※ Parameter와 Argument

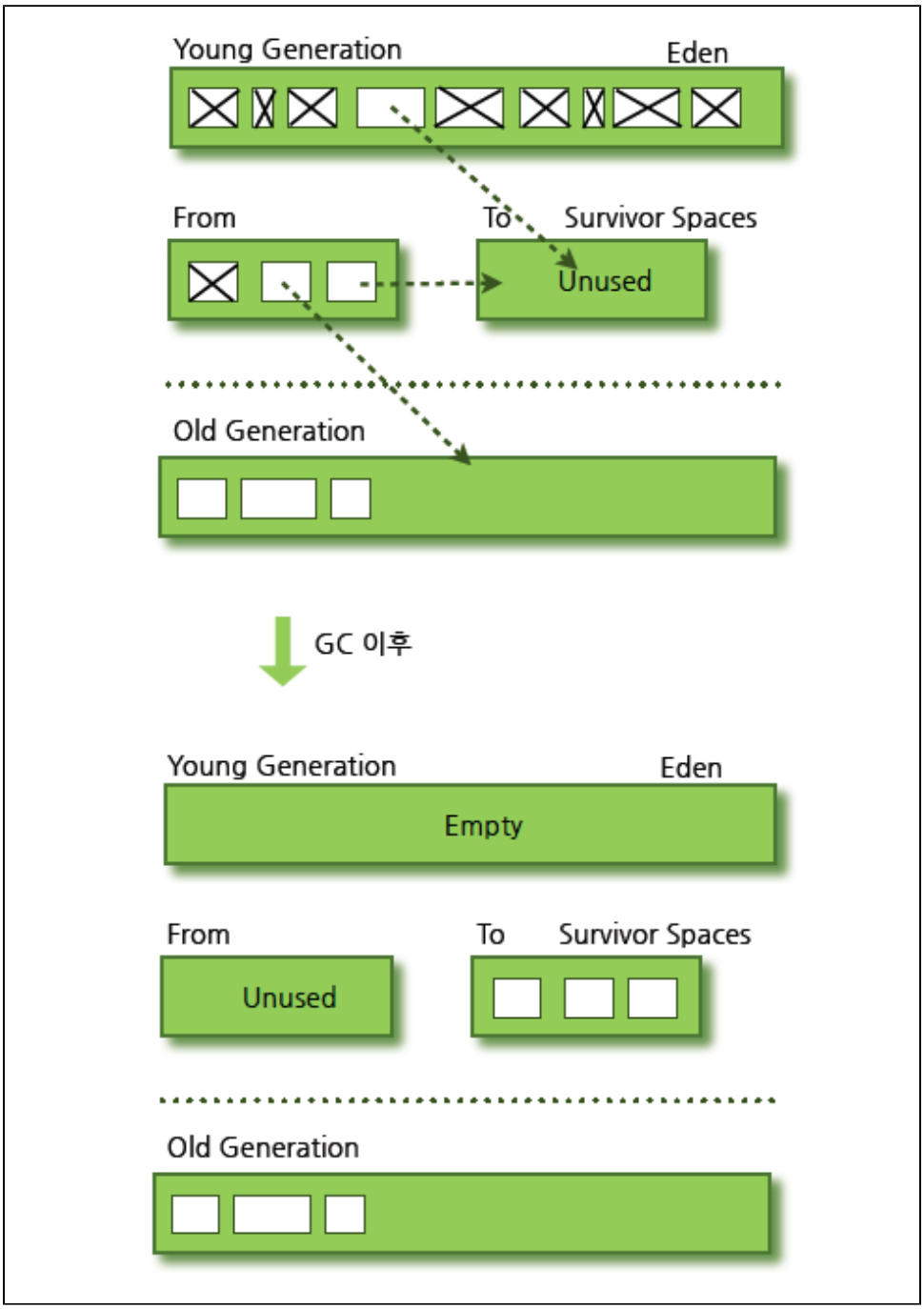
```
public void method(Integer i, String s, Object o){

}


public static void main(String[] args){
    method(1, "1", new Object());
}
```

(3) Heap Area

- **Young Generation**
  - 새로생성한 객체의 대부분이 위치하는 영역으로 대부분의 객체가 금방 접근 불가능 상태가 되므로 매우 많은 객체가 생성되었다가 사라진다
    Minor GC가 발생하는 영역이며 Eden영역, Survivor1, Survivor2 영역으로 나뉜다

  - Eden 영역과 두개의 Survivor 영역의 동작
    - 새로 생성한 대부분의 객체는 Eden 영역에 위치
    - 이곳에서 GC가 한 번 발생한 후 살아남은 객체는 Survivor(1) 영역 중 하나로 이동 (Survivor1)
    - Eden 영역에서 GC가 발생하면 이미 살아남은 객체가 쌓인 Survivor(1) 영역으로 계속 쌓인다
    - 하나의 Survivor(1) 영역이 가득 차서 GC가 발생하면 살아남은 객체를 다른 Survivor(2) 이동한다 (Survivor1은 비어있다)

- 위 과정을 반복하여 계속해서 살아남은 객체는 Old 영역으로 이동



- **Old Generation**
  - Old 영역은 기본적으로 데이터가 가득차면 GC를 실행하며 Garbage Collector에 따라 처리 절차가 다르다
  - Serial GC
  - Parallel Gc
  - Parallel Old GC (Parallel Compacting GC)
  - Concurrent Mask & Sweep GC (CMS)
  - G1 (Garbage First) GC

- **Permanent Generation**

- Class의 Meta 정보(Package에 따른 path 정보)
- Method의 Meta 정보
- Static Object
- 상수화 된 String Object
- Class와 관련된 배열 객체의 Meta 정보
- JVM 내부적인 객체들과 JIT 컴파일러의 최적화 정보

※ java.lang.OutOfmemoryError : PermGen space ?

프로젝트 크기가 커짐에 따라 위 정보들이 늘어나고 Application 구동시 필요한 PermGen이 늘어남
→ MaxPermSize를 JVM 옵션에 명시하여 적절히 늘려주어야 한다
→ Java8에서는 고질적인 perm gen space error을 해결하기 위해 perm 영역을 없앰

2. Garbage Collector

※ Garbage Collector를 설명하기에 앞서...

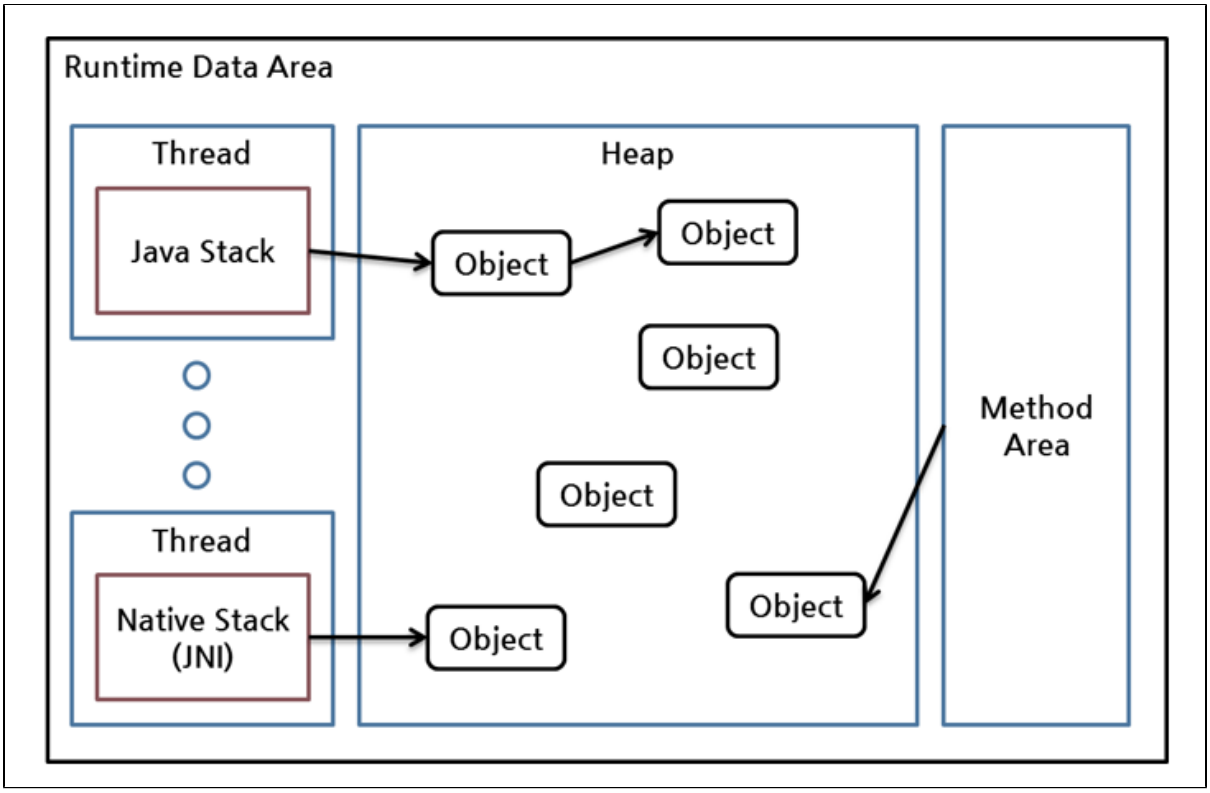
1) GC가 되는 대상 메모리 영역은?

2) GC가 되는 대상의 판단은?

3) Java의 type 종류는? (Primitive, Reference type)
→ Reference Type의 종류는?

① GC와 Reachability

Reachable vs Unreachable 그리고 Root set



Heap 내의 객체 참조 종류

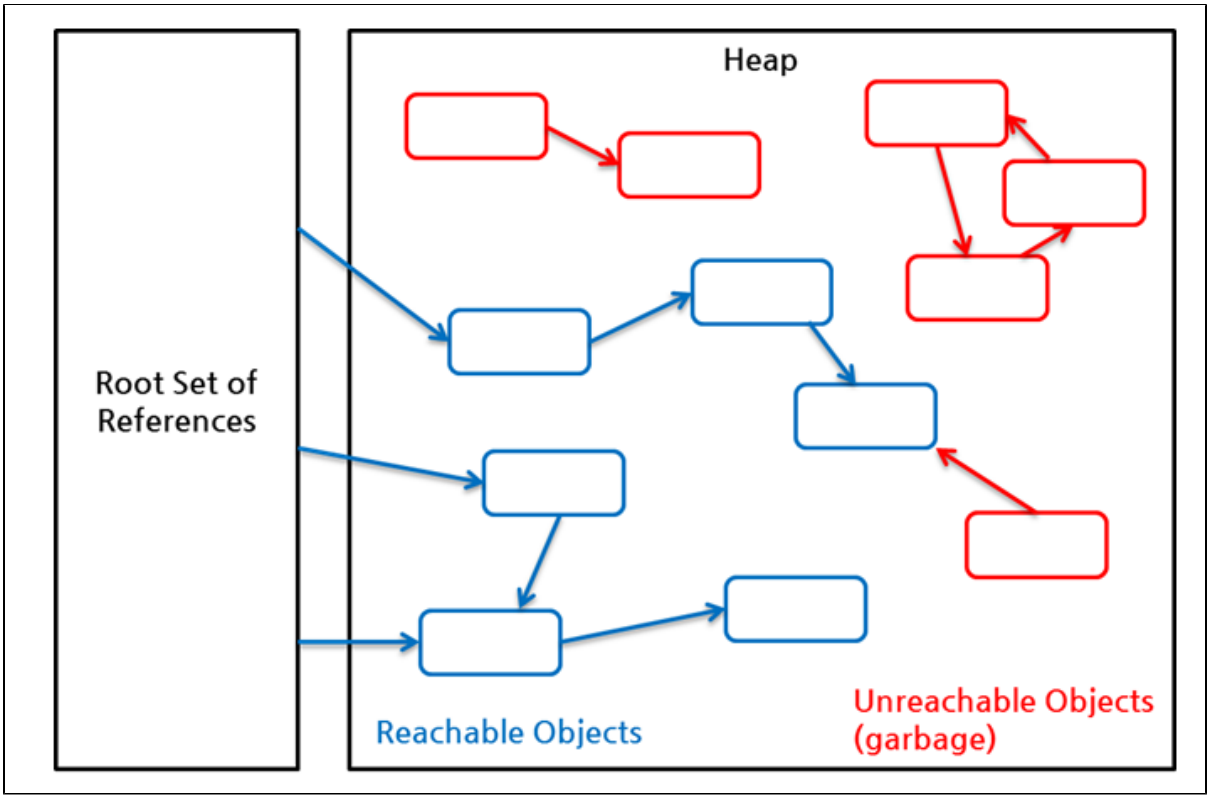
- Heap 내의 다른 객체에 의한 참조
- Stack에 의한 참조(Method 실행 시 사용하는 지역변수, 파라미터들에 의한 참조)
- Native stack, 즉 JNI에 의해 생성된 객체에 대한 참조
- 메소드 영역의 정적 변수에 의한 참조

1을 제외한 나머지 3개가 Root set으로 Reachability를 판단하는 기준이 된다

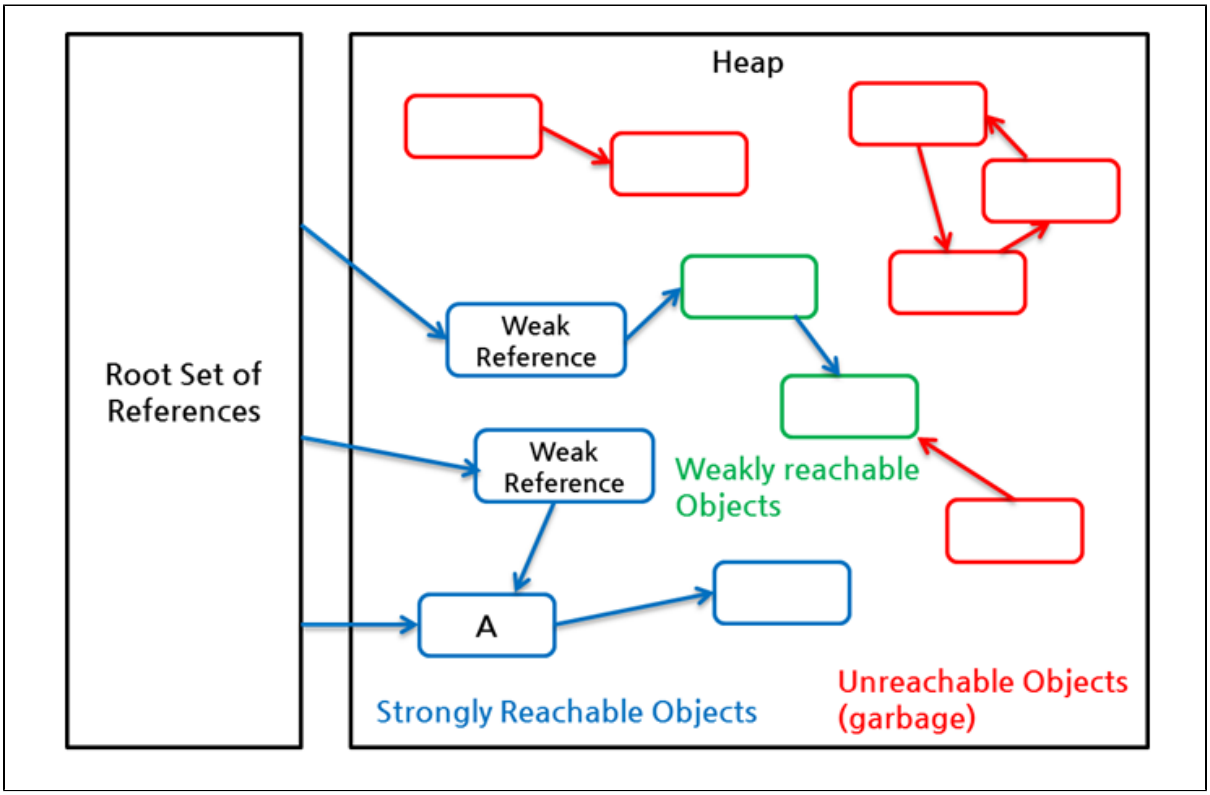
② Reference와 Reachability

reference object란 SoftReference, WeakReference, PhantomReference 3가지 클래스에 의해 생성된 객체를 일컫는다

일반적인 Reachable과 Unreachable 그림



WeakReference가 존재하는 경우 Reachable과 Unreachable



GC가 동작할 때 Unreachable 객체 뿐 아니라 Weakly reachable도 가비지 대상으로 간주 되어 메모리에서 해제 됨.

→ Root set으로 부터 시작된 참조 사슬에 있지만 반드시 항상 유효할 필요가 없는 LRU 캐시와 같은 임시 객체를 저장하는 구조에 유리

1) **Strongly reachable**: Root set으로 부터 시작해서 어떤 reference object도 중간에 끼지 않은 상태로 참조 가능한 객체
→ 객체까지 도달하는 여러 참조 사슬 중 reference object가 없는 사슬이 하나라도 있는 객체

2) **Softly reachable**: Strongly reachable 객체가 아닌 객체중에 weak reference, phantom reference 없이
soft reference 만 통과하는 참조 사슬이 하나라도 있는 객체
→ 힙에 남아있느 메모리 크기와 해당 객체의 사용 빈도에 따라 GC 여부가 결정 됨
→ Weakly reachable 객체와 달리 GC가 동작할 때마다 회수되지 않으며 자주 사용될수록 더 오래 살아남는다

3)**Weakly reachable**: strongly reachable 객체도 softly reachable 객체도 아닌 객체 중에서, phantom reference 없이
weak reference만 통과하는 참조 사슬이 하나라도 있는 객체
→ 특별한 정책에 따라 GC 여부가 되는 Softly reachable 객체와 달리 GC 수행 시마다 회수 대상이 된다
→ GC가 언제 객체를 회수할 지는 GC 알고리즘에 따라 다르므로 GC가 수행 될 때마다 반드시 메모리까지 회수된다고 보장하지 않음
→ LRU 캐시와 같이 Softly reachable 객체보다는 Weakly reachable 객체가 유리하여 대체로 WeakReference를 사용한다

4) **Phantomly reachable**: strongly reachable 객체, softly reachable 객체, weakly reachable 객체 모두 해당되지 않는 객체.
이 객체는 파이널라이즈(finalize)되었지만 아직 메모리가 회수되지 않은 상태이다.

5) **Unreachable**: root set으로부터 시작되는 참조 사슬로 참조되지 않는 객체

(1) Stop the world

Garbage Collection이 진행되는 동안 GC Thread를 제외한 Application의 모든 Thread는 멈춘다

- **Minor GC**
  - Young generation에서 발생한 GC를 일컫는다
  - 비교적 사이즈가 작은 영역에 대한 GC이므로 적은 시간이 소요된다
    → Stop the world의 시간이 적다
  - 자주 발생한다
  - Young generation 대부분의 연속된 영역을 모두 비우므로 Compacting 시간이 적다
- **Major GC(Full GC)**
  - Old Generation에서 발생한 GC를 일컫는다
  - 비교적 크기가 큰 영역에 대한 GC 이므로 많은 시간이 소요된다
  - 적게 발생한다
  - 듬성 듬성 비연속적 영역을 비우므로 파편화가 발생하면 Compacting 시간이 길다

※ 짧게 자주 멈추는것 vs 길게 한 두번 멈추는 것?

(2) Generational GC

- David ungar의 논문 'Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm'
  → 대부분의 객체는 일찍 죽는다라는 가설

→ 실제 통계로도 생성된 객체의 98%가 곧바로 Garbage 상태가 된다

- 이러한 가설을 바탕으로 설계된 것으로 Heap을 Young Generation 영역과 Old Generation 영역으로 나눈 뒤 Young Generation 영역을 주기적으로 청소하고 오래 살아남은 객체를 Old Generation 영역으로 내보낸다는 개념

- Heap 전체를 매번 뒤져서 청소(+Compacting)하지 않고 주로 Young Generation을 청소하다가 Old Generation 공간이 부족한 경우에 Old 영역을 청소하자

(3) Garbage Collector 종류

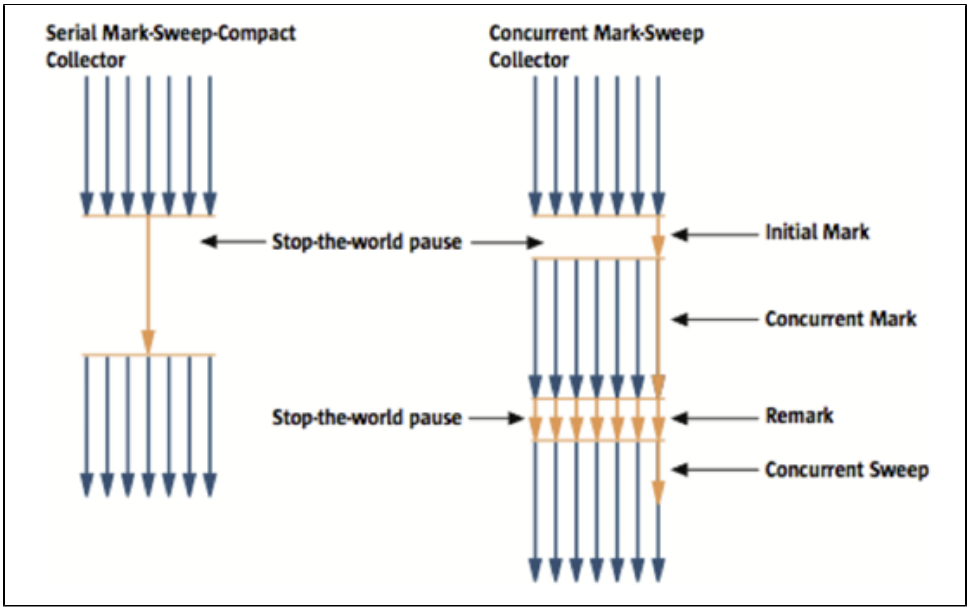
기본적으로 Minor GC는 Garbage Collector의 종류에 상관없이 모두 Stop the world가 발생

- **Serial GC**
  - Mark-Sweep-Compaction 알고리즘 사용
  - 가장 단순한 GC
  - 주로 32bit JVM에서 동작하는 Single thread application에 사용 됨
  - Minor GC와 Full GC에서 모두 Stop the world가 발생
  - GC에 수행되는 Thread가 하나
  - 적은 메모리와 CPU 코어 개수에 적합

- **Parallel GC (Throughput GC)**
  - Serial GC와 알고리즘은 동일
  - 병렬화 된 GC
  - 주로 64bit JVM에서 동작하는 application에 사용
  - Minor GC와 Full GC에서 모두 멀티 스레드를 사용하지만 여전히 Stop the world 발생
  - 메모리가 충분하고 코어의 개수가 많을 때 유리

- **Parallel Old GC(Parallel Compacting GC)**
  - 이전의 Parallel GC와 비교하여 Old 영역의 GC 알고리즘이 다름
  - Mark-Summary-Compaction 단계를 거침
  - Summary 단계에서 GC를 수행한 영역에 대해 별도로 살아있는 객체를 식별한다

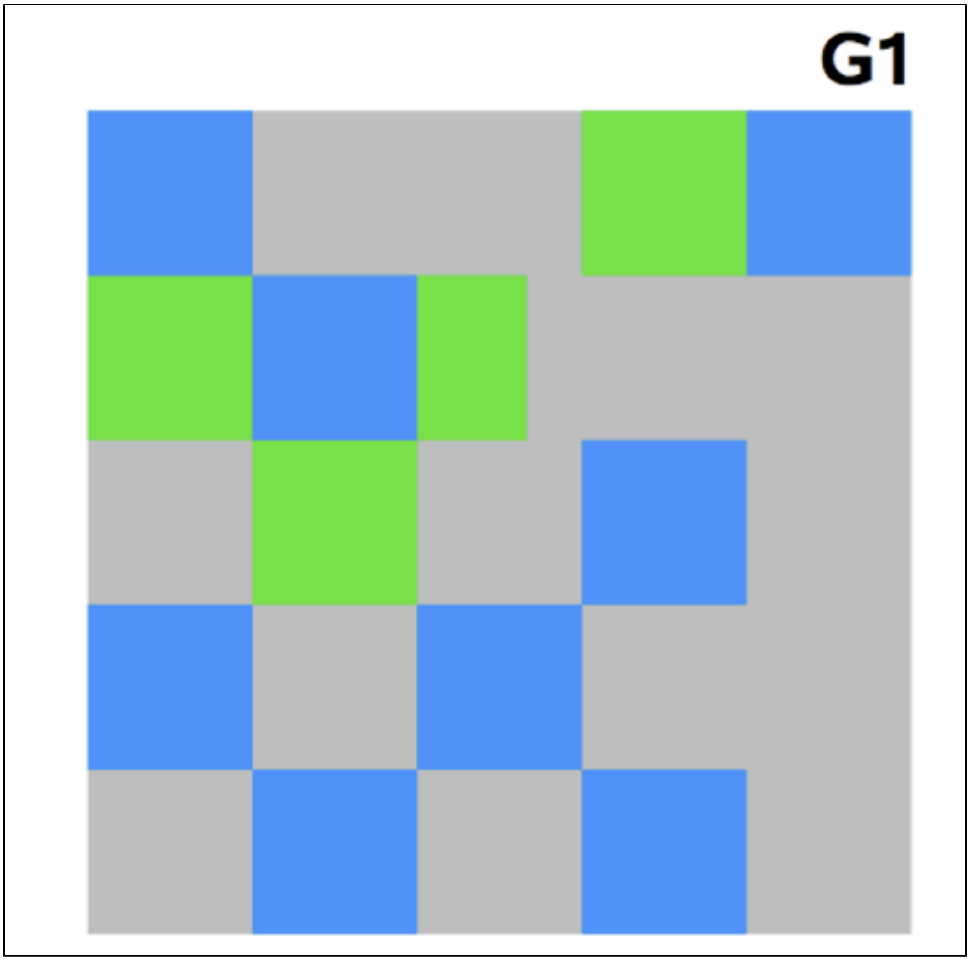
- **Concurrent Mask Sweep(CMS) Collector**



- Full GC의 Stop-the-world 상태를 줄일 수 있을까라는 고민에서 출발
- Parallel GC와 동일하게 멀티스레드로 Minor GC 수행 (알고리즘이 다름)
- Full GC에서 거의 Stop the world가 발생하지 않음 (low-latency)

    - 백그라운드 스레드를 통해 Old Generation에 살고 있지만 쓸모없는(참조가 없는) 객체를 지속적으로 제거
    -
        - 최초 Initial Mark 단계에서 클래스 로더에서 가장 가까운 객체 중 살아있는 객체만 탐색 (멈추는 시간 매우 짧음)
        - Concurrent Mark 단계에서 방금 살아있다고 확인한 객체에서 참조하는 객체를 따라가며 확인 (다른 Work thread와 동시에 동작)
        - Remark 단계에서는 Concurrent Mark 단계에서 새로 추가되거나 참조가 끊긴 객체를 확인
        - Concurrent Sweep 단계에서는 쓰레기를 정리 (다른 Work thread와 동시에 동작)
- 백그라운드에서 동작하는 스레드 때문에 CPU 리소스가 많이 잡아 먹힘
- 지속적인 백그라운드 GC로 인해 Old Generation 중간 중간이 비어버림
  → 메모리의 파편화가 발생함
  → 가용 메모리가 점점 줄어들음
  → Old Generation 영역을 싱글스레드로 정리하며 Compaction 작업을 처리
  → Stop the world가 매우 길게 발생
- Compaction 작업이 얼마나 자주 발생하며, 오래 수행되는지 확인이 필요

- **G1 (Garbage First) GC**
    - Heap area의 Young Generation, Old Generation 구조를 없앰



- Heap에 영역(Region)이라는 개념을 도입하여 여러개의 Region으로 나누고 Young Generation과 Old Generation으로 나눔
    - 특정 영역을 Young, Old로 지정하지 않음, 언제든 Young, Old가 될 수 있음
- Minor Gc는 Paraller GC
- Full GC는 CMS와 같이 백그라운드 스레드로 진행 됨
    - CMS와 차이점은 중간 중간 쓸모없는 객체를 정리하는 것이 아니라 Region을 통째로 정리함
      → CMS에서 문제였던 메모리 파편화가 발생하지 않음 (백그라운드이므로 CPU 리소스는 여전히 사용)
    - Old 영역의 Heap 사용량이 특정 Threshold 값을 넘어서면 실행 됨
- CMS GC를 대체하기 위해 만들어진 것으로 그 어떤 GC와 비교하여 성능이 빠름
- Parallel GC와 CMS GC의 철충안에 해당하며 준수한 throughput, latency를 제공 (high-throughput, low-latency)
- Heap이 많이 필요한 경우 G1 GC를 사용하는 것이 유리

3. Thread Local

(1) 공유 변수로 인한 문제

Multi thread가 아래의 클래스로부터 생성된 singleton object를 공통으로 사용한다고 가정 할 때 발생하는 문제

```
public abstract class AbstractMessageService {
        private LocaleService localeService;
```

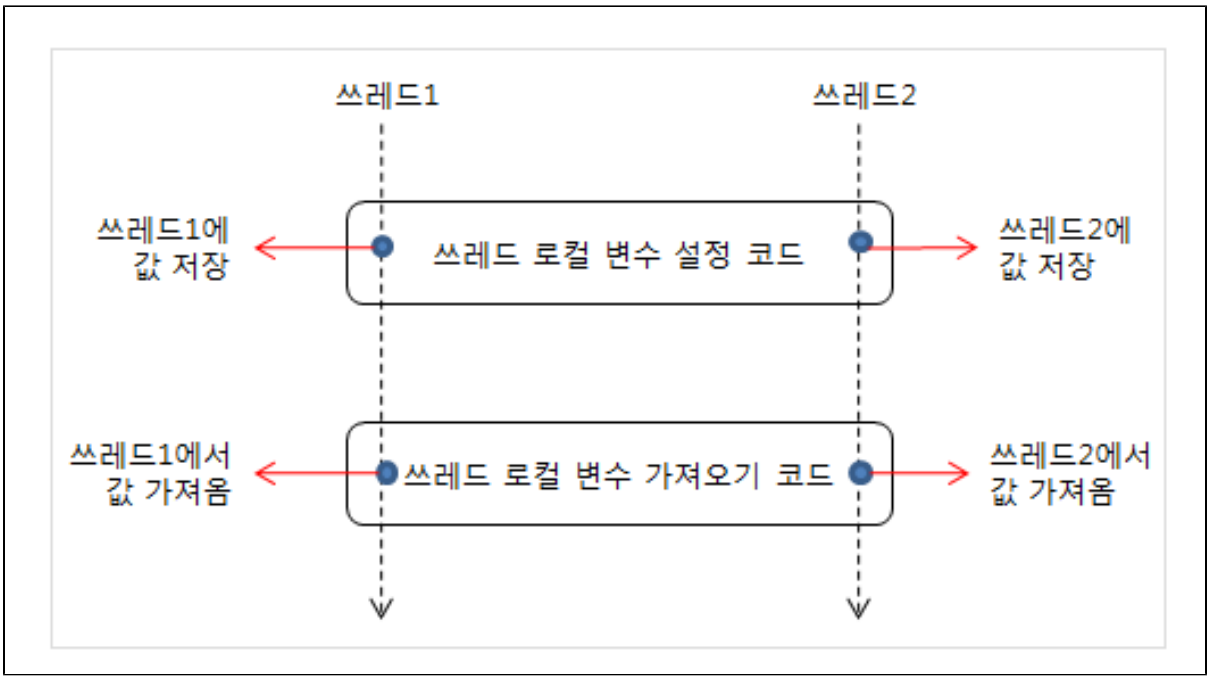
```
        public AbstractMessageService(LocaleService localeService){
                this.localeService = localeService;
        }


        public Locale getLocale(User user){
                return this.localeService.getLocale(user);
        }


        public String getLocalizedMailTemplate(String mailTemplateId, User user){
                return getLocalizedMailTemplateHeader(mailTemplateId, user) + "\n" +
getLocalizedMailTemplateBody(mailTemplateId, user);
        }
        public Locale getLocale(User user){


        }
        public abstract String getLocalizedMailTemplateHeader(String mailTemplateId, User user);
        public abstract String getLocalizedMailTemplateBody(String mailTemplateId, User user);
}


@Service
public class MessageServiceImpl extends AbstractMessageService {

        @Override
        public String getLocalizedMailTemplateHeader(String mailTemplateId, User user) {
                Locale locale = super.getLocale(user);
        }

        @Override
        public String getLocalizedMailTemplateBody(String mailTemplateId, User user) {
                Locale locale = super.getLocale(user);
        }
}


/**
*   locale service    caching
*/
@Service
public class AdvancedMessageServiceImpl extends AbstractMessageService {
        private Locale locale;


        @Override
        public Locale getLocale(User user){
                return this.locale == null
                                ? super.getLocale(user)
                                : this.locale;
        }

        @Override
        public String getLocalizedMailTemplateHeader(String mailTemplateId, User user) {
                Locale locale = this.getLocale(user);
        }

        @Override
        public String getLocalizedMailTemplateBody(String mailTemplateId, User user) {
                Locale locale = this.getLocale(user);
        }
}
```

(2) Thread Local 개념 및 사용법

1) 개념

특정 Thread가 실행하는 모든 코드에서 그 Thread에 설정 된 값을 사용할 수 있도록 한다

→ Thread의 실행 콘텍스트 전역에 걸쳐 유효한 공유변수



2) 기본 사용법

- ThreadLocal 객체를 생성한다
- ThreadLocal.set() 메소드를 통해 현재 thread의 local 변수에 값을 저장한다
- ThraedLocal.get() 메소드를 통해 현재 thread의 local 변수 값을 읽어온다
- ThreadLocal.remove() 메소드를 통해 thread의 local 변수에 설정된 값을 삭제한다

3) 주 활용처

- 사용자 인증정보 전파: Spring security에서 사용자 인증 처리는 ThreadLocal을 통해 구현되어있음
  - Spring security의 SecurityContextHolder 참조
  - Restful API에서 사용자 인증 토큰을 interceptor에서 파싱 후 ThreadLocal에 담아두어 Request context 전역에서 사용 가능
- 트랜잭션 컨텍스트 전파: Spring의 TransactionManager는 ThreadLocal을 통해 트랜잭션의 컨텍스트를 관리함
- Locale 처리: Spring의 DispatcherServlet은 LocaleResolver에 의해 파악한 Locale 정보를 LocaleContextHolder에 저장하여 해당 정보는 Request context 전역에서 참조 가능함
- Thread safe 해야하는 값의 보관

※ 주의사항

Thread pool 환경에서 ThreadLocal을 사용하는 경우 ThreadLocal에 보관된 데이터의 사용이 끝나면
반드시 해당 데이터를 삭제해주어야 한다.

그렇지 않은 경우 **올바르지 않은 데이터를 참조**하거나 **장기적인 메모리 누수**가 발생할 수 있다

속성정리

학습내용 정리

- 객체지향 3요소 5원칙
- Web server와 Web Application Server
    - 나누는 이유는?
    - Web Descriptor (Deployment Descriptor)
- Spring의 3대 핵심기술
    - Spring은?
- Servlet의 동작과 생명주기
- Servlet과 JSP의 차이점?
- JSP & Servlet Model 1, 2
- MVC Pattern과 Spring MVC
- JVM 구조
- Heap 구성
- Garbage Collector 동작
- Thread Local

Java 8 추가 내용

Optional

- Java에서 대부분 발생하는 예외는 NPE
    - **Null을 피하기 위한 일반적인 코드**

```
Cart cart = response.getCart();
if (cart != null) {
  Product product = cart.getProduct();
  if (product != null) {
    System.out.println(product.getName());
  }
}
```

    Optional을 이용한 Null을 피하는 코드

```
Optional.ofNullable(response.getCart()).ifPresent(c -> {
  Optional.ofNullable(c.getProduct()).ifPresent(p -> System.out.println(p.getName()));
});
```

※ Lambda와 Method reference

Lambda

```
String [] strings = new String [] {"6", "5", "4", "3", "2", "1" };
List<String> list = Arrays.asList(strings);
list.forEach(x -> System.out.println(x));
```

Method reference

```
String [] strings = new String [] { "6", "5", "4", "3", "2", "1" };
List<String> list = Arrays.asList(strings);
list.forEach(System.out::println);
```

Stream

- Java의 Stream은 컬렉션, 배열 등의 반복처리를 추상화하여 내부 반복으로 처리함
  - 외부반복과 내부반복
  - Collection 인터페이스는 사용자가 직접 요소를 반복함 → **외부반복**
    - *방바닥에 장난감 있지?*

      → 네

      → 주워 담아, 또 장난감이 있니?

      .... 반복
  - Stream 라이브러리는 반복을 알아서처리하고 결과 스트림 값을 어딘가에 저장해줌 → **내부반복**
    - *방바닥에 있는 장난감 다 주워담아.*
- 스트림의 구조
  - 스트림 생성
    - Stream.of(), Collection.stream(), IntStream.range()
  - 중개 연산
    - Stream.filter(), Stream.map(), Stream.flatMap(), Stream.peek(), Stream.sorted(), Stream.limit(), Stream.distinct, Stream.skip()
  - 최종 연산(종단 연산)
    - Stream.count(), Stream.min(), Stream.max(), Stream.sum()
    - Stream.collect(), Stream.groupingBy(), Stream.partitioningBy(), Stream.reduce()
      - 다양한 Collector가 존재
    - Stream.forEach(),
    - Stream.noneMatch(), Stream.anyMatch(), Stream.allMatch(), Stream.findFirst(), Stream.findAny()
  - Collections같은 객체 집합.스트림생성().중개연산().최종연산();
- Lazy 연산을 지원한다
  - filter와 map은 연산되지 않고 있다가 실제 스트림의 최종연산이 수행 될 때 적용된다

Optional을 이용한 Null을 피하는 코드

```
Stream<String> a = names.stream().filter(x -> x.contains("o")).map(x-> x.concat("s"));
a.forEach(x -> System.out.println(x));
```

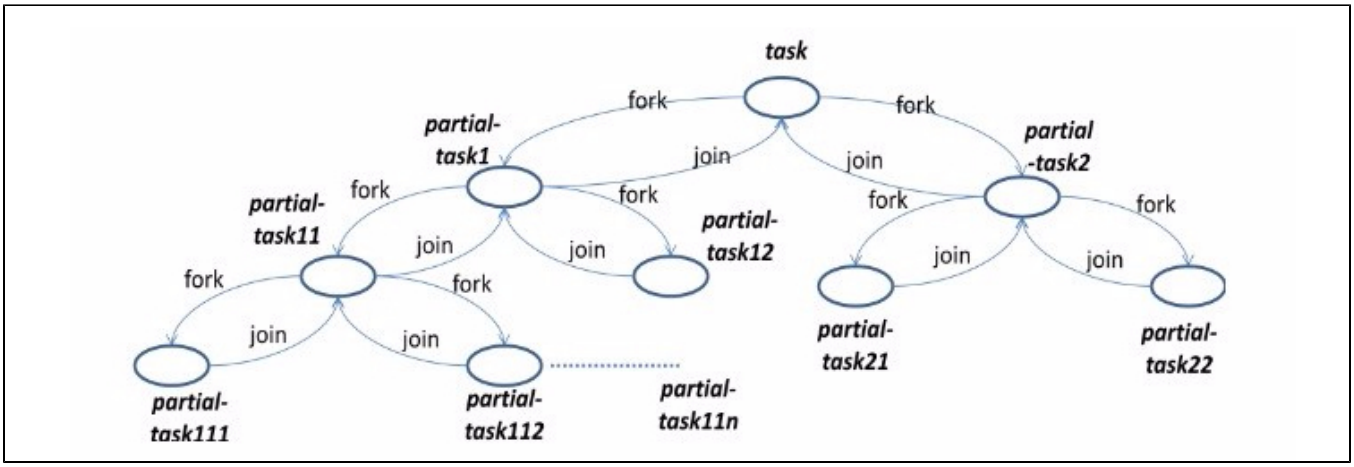
- Stream은 재사용이 불가능
- 손쉬운 병렬처리
    - Stream 생성 시 stream()이 아닌 parallelStream()을 통해 병렬 스트림을 만들 수 있음
    - 내부적으로 ForkJoinPool을 사용함
    - 람다를 사용하므로 Thread의 context가 보장되므로 동기화 신경쓸 필요가 없음
    - 항상 성능측정을 통해 비교 필요
        - 항상 Task는 균등하게 나뉘어야 함

          → 균등히 나뉠 수 있도록 ForkJoinPool의 커스터마이징 등
        - 병렬로 처리되는 작업이 독립적이지 않다면, 수행 성능에 영향이 있을 수 있음
            - sorted(), distinct() 등의 연산이 있다면 내부적으로 synchronized 되어 오히려 성능에 악영향
    - 언제 써야할까?
        - ForkJoinPool을 통해 분할이 균등히 잘 이뤄질 수 있는 데이터 구조
        - 작업이 독립적이며 CPU 사용이 높은 작업
            - parser, crawling,
        - J2EE Container 환경에선 절대 사용하지 말 것

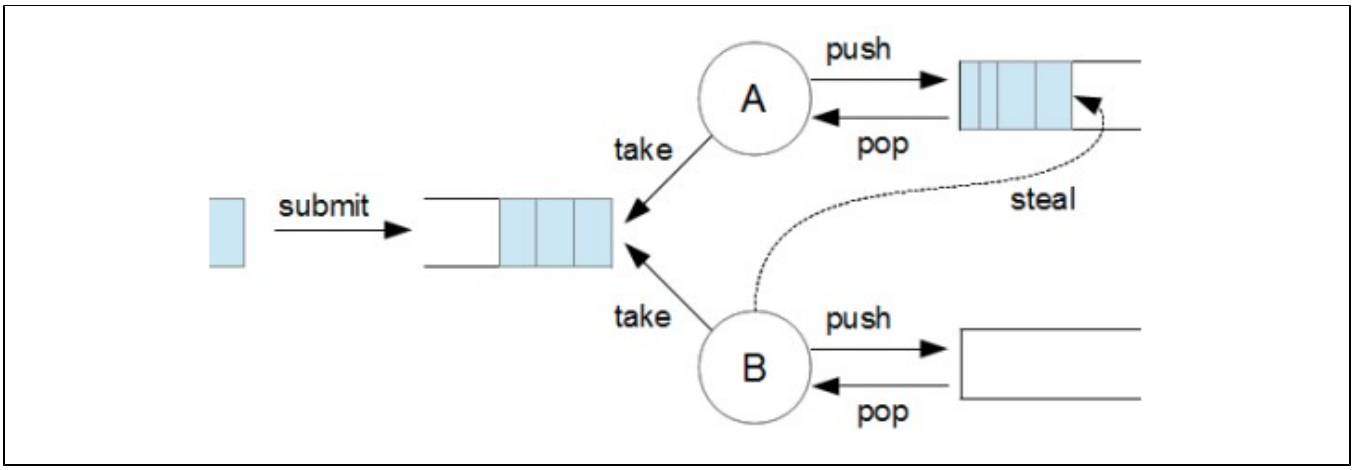
※ 동시성(Concurrency)과 병렬성(Parallelism)

- 동시성
    - Single core에서 Multi thread를 동작시키기 위한 방식으로 멀티태스킹을 위해 여러개의 Thread가 번갈아가면서 실행
- 병렬성
    - Multi core에서 Multi thread를 동작시키는 방식으로 한 개 이상의 Thread를 포함하는 각 core들이 동시에 실행
        - 데이터 병렬성
            - 전체 데이터를 쪼개 서브 데이터를 만든 뒤 병렬처리하여 작업을 빠르게 수행하는 것
            - Java8의 Parallel stream
        - 작업 병렬성
            - 서로 다른 작업을 병렬처리 하는 것
            - ex) 웹 서버는 개별 스레드에서 각 요청을 병렬로 처리한다

ForkJoinPool 동작



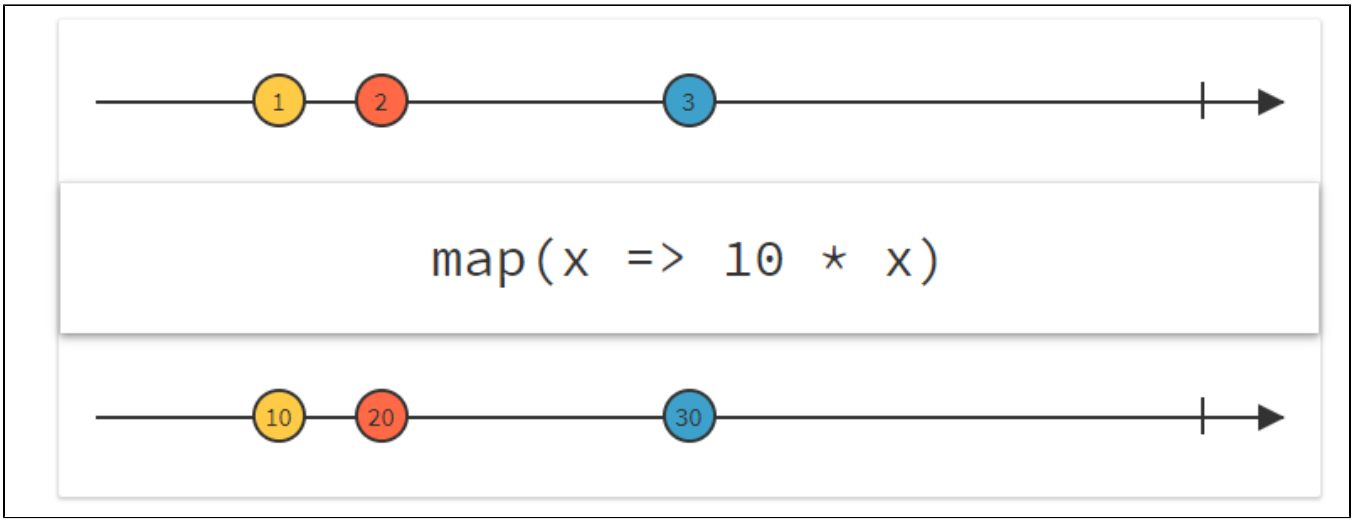
→ 정해진 threshold까지 분할하여 가장 작은 일부터 병합.



→ 모든 Thread들이 개별 task queue를 가지며 만약 task가 없다면 다른 Thread에 할당된 task를 Steal함.

※ .map()과 .flatMap()의 차이

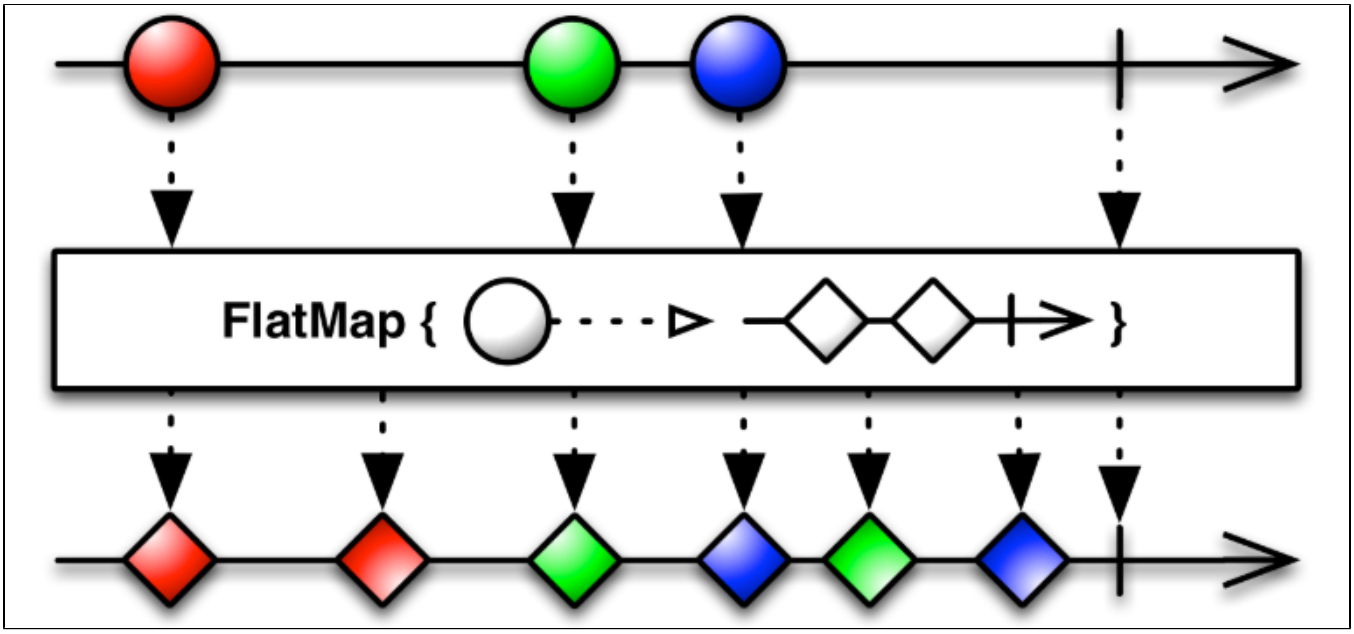
.map()



.map()은 단일 스트림의 원소를 매핑하여 그 값을 다시 스트림으로 반환하는 중간연산이다.

→ Stream이라는 Container가 담고있는 원소의 차원이 바뀐다

.flatMap()



.flatMap()은 여러개의 스트림을 하나의 스트림으로 합치는 역할을 하며 flat 하게 펴낸 후 다시 스트림으로 반환하는 중간연산이다.

→ flagMap()에서 반환한 Stream을 현재 차원의 Stream에 합병한다

01. Google fonts - Early access crawling

CompletableFuture, Stream 종합 예제코드

```
@Slf4j
@Service
public class FontCrawlingService {

        @Value("${app.thread_pool_size}")
        private int threadPoolSize;

        @Value("${app.default_save_root_path}")
        private String saveRootPath;

        private ExecutorService executor;
        private RestTemplate restTemplate;

        @PostConstruct
        public void init() {
                executor = Executors.newFixedThreadPool(threadPoolSize);
                restTemplate = new RestTemplate();
                restTemplate.getMessageConverters().add(new ByteArrayHttpMessageConverter());
        }

        /**
         * <code>targetUrls</code>
         * <p>
         * targetUrls  font-face   css
         * HttpMethod.GET
         *
         * @param targetUrls    url
         */
        public void crawling(List<String> targetUrls) {
                List<CompletableFuture> futures = targetUrls.stream()
                                .map(this::getDocumentFromUrl)
                                .filter(Optional::isPresent)
```

```
                                .map(document -> document.get().text())
                                .flatMap(document -> Stream.of(document.split("url")))
                                .map(mapped -> mapped.split("\\(")[1])                          //(http://fonts.
gstatic.com....) '('
                                .map(mapped -> mapped.split("\\)")[0])                          // http://fonts.
gstatic.com....) ')'
                                .filter(this::isValidFontUrl)
                                .map(url ->
                                                CompletableFuture.supplyAsync(() -> findFont(url), executor)
                                                                .thenAccept((res) -> saveFont(url, res))
                                ).collect(Collectors.toList());

                futures.forEach(CompletableFuture::
join);                                                  //Blocking to all completable futures
                log.info("====================finished==========================");
        }



        /**
         * jsoup  <code>url</code>
         * <p>
         * Jsoup connect       IOException
         * Wrapping
         *
         * @param url  URL
         * @return Optional of Document
         */
        private Optional<Document> getDocumentFromUrl(String url) {
                try {
                        return Optional.ofNullable(Jsoup.connect(url).get());
                } catch (IOException e) {
                        return Optional.empty();
                }
        }

        /**
         * split   <code>parsed</code>  Font url
         * Early access  font url
         * <p>
         * https://xxx
         * //fonts.xxx
         *
         * @param parsed
         * @return boolean
         */
        private boolean isValidFontUrl(String parsed) {
                return parsed.contains("https://") || parsed.contains("//fonts.");
        }

        /**
         * <code>url</code>  font
         *
         * @param url font url
         * @return ResponseEntity
         */
        private ResponseEntity<byte[]> findFont(String url) {
                if(!url.startsWith("https"))
                        url = "https:" + url;

                log.info("find font from -> " + url);
                return restTemplate.getForEntity(url, byte[].class);
        }

        /**
         * <code>url</code>  font<code>res</code>
         *
         * @param url   url     .
         * @param res   Response
         */
        private void saveFont(String url, ResponseEntity<byte[]> res) {
```

```
                log.info("save file call -> " + url);

                String directoryPath = getDirectoryPath(url);
                if(!Files.exists(Paths.get(directoryPath))) {
                        log.info("mrdirs for path [" + directoryPath + "] result -> " + new File(directoryPath).
mkdirs());
                }

                try {
                        Files.write(Paths.get(directoryPath + "/" + getFilName(url)), res.getBody());
                } catch (IOException e) {
                        log.error("   Exception [" + e.getMessage() + "]", e);
                }
        }

        /**
         * url  font
         *
         *  url
         * https://fonts.gstatic.com/s/notosansjp/v14/-F62fjtqLzI2JPCgQBnw7HFYwQgP-FVthw.woff2
         *
         *  String
         * <code>saveRootPath</code>/s/notosansjp/v14
         *
         * @param url   url
         * @return
         */

        private String getDirectoryPath(String url) {
                String fontURI = url.split(".com")[1];    //fontURI maybe /s/notosansjp/v14/-
F62fjtqLzI2JPCgQBnw7HFYwQgP-FVthw.woff2
                String[] splits = fontURI.split("/");

                return saveRootPath + "/" + StringUtils.join(Arrays.asList(splits).subList(0, splits.length -
1), '/');
        }

        /**
         * <code>url</code>
         * <p>
         *   url  [filename.woff2] .
         * https://test.abc.com/parent/child/filename.woff2
         * <p>
         *     "default"
         *
         * @param url   url
         * @return
         */
        private String getFilName(String url) {
                String[] splits = url.split("/");
                if(splits.length > 0)
                        return splits[splits.length - 1];
                else
                        return "default" + url.substring(1, 10);
        }
}
```