Wrapper Class

자바에는 기본 자료형(Primitive data type)이 있는데 그에 대한 클래스 표현인 Wrapper Class가 존재한다.

- 기본 타입 : int, long, float, double, boolean ...
- 래퍼 클래스 : Integer, Long, Float, Double, Boolean ...

박싱과 언박싱

- 박싱 : 기본 데이터 타입에 대응하는 wrapper클래스로 만드는 동작

```
int i = 10;
Integer num = new Integer(i);   // Boxing
```

- 언박싱 : Wrapper 클래스에서 기본 타입으로 변환

```
Integer num = new Integer(10);
int i = num.intValue();         //unBoxing
```

오토 박싱

편의성을 위해 오토 박싱과 오토 언박싱이 제공된다. (JDK 1.5 이상)

```
int i = 10;
Integer num = i; // Auto Boxing

Integer num = new Integer(10);
int i = num;        // Auto unBoxing
```

하지만 이러한 오토 캐스팅은 100만건 기준 약 5배의 성능차이가 나므로 불필요한 오토 캐스팅이 일어나는지 확인하는 습관을 가져야 한다.

```
ArrayList<Integer> lists = new ArrayList<>();
int i = 10;
lists.add(i);

Integer i = new Integer(10);
lists.add(i);
```

문자열 클래스

String, StringBuffer, StringBuilder

- String : 불변 / 동기화X
- StringBuffer : 가변 / Synchronized 가능(Thread-safety)
- StringBuilder : 가변 / Synchronized 불가능

String 특징

- new 연산을 통해 생성된 인스턴스의 메모리 공간은 변하지 않는다.(Immutable)
- Garbage Collector로 제거되어야 한다.
- 문자열 연산시 새로 객체를 만드는 Overhead가 발생한다.
- 객체가 불변하므로, Multi-Thread 환경에서 동기화를 신경 쓸 필요가 없다.

> String Class : 문자열 연산이 적고, 조회가 많은 멀티쓰레드 환경에서 사용하기 좋다.

StringBuffer, StringBuilder 특징

차이점

- StringBuffer : Thread-safe함
- StringBuilder : Thread-safe하지 않음

공통점

- new 연산으로 클래스를 한번만 만든다.(Mutable)
- 문자열 연산시 새로 객체를 만들지 않고 크기를 변경시킨다.
- StringBuilder와 StringBuffer 클래스의 메서드가 동일하다.

> 문자열 연산이 많을 때 쓰기 좋다.
>
> StringBuffer Class : Multi-Thread 환경
>
> StringBuilder Class : Single-Thread 또는 Thread를 신경쓰지 않는 환경

Call by value & Call by reference

Call by value

> 값에 의한 호출

함수가 호출될 때 메모리 공간 안에서는 함수를 위한 별도의 임시공간이 생성된다. (종료시 반환)

Call by value 호출 방식은 함수 호출 시 전달되는 변수 값을 복사해서 함수 인자로 전달한다.

이 때 복사된 인자는 함수 안에서 지역적으로 사용되기 때문에 **local value속성**을 가진다.

```java
package Practice;

import java.util.*;

public class Just_Practice {
    public static void main(String[] args) {
        int a = 0;
        test(a);
        System.out.println(a);
    }

    static void test(int a) {
        a = 10;
    }
}
```

```
<terminat
0
```

> 따라서, 함수 안에서 인자 값이 변경되더라도, 외부 변수 값은 변경되지 않는다.

Call by reference

> 참조에 의한 호출

함수 호출 시 인자로 전달되는 변수의 레퍼런스를 전달한다.

> 함수 안에서 인자 값이 변경되면, 인자로 전달된 객체의 값도 변경된다.

자바의 경우, 항상 Call by value로 값을 전달한다.

C/C++과 같이 변수의 주소값을 가져올 방법이 없다. 그러니 주소값을 넘기는 방식도 존재하지 않는다.

자바에서는 Call by value 방식을 수행할 때, 값을 넘겨받은 메소드에서 인자 값을 복사하여 새로운 지역 변수에 저장한다. 따라서 변수의 값이 변경되지 않았던 것이다.

하지만 객체를 인자로 넣어준다면 값이 변하게 된다.

```java
package Practice;

import java.util.*;

public class Just_Practice {
    static class Obj{
        int num;
        String str;

        public Obj(int num, String str) {
            super();
            this.num = num;
            this.str = str;
        }

        @Override
        public String toString() {
            return "Obj [num=" + num + ", str=
        }
    }

    public static void main(String[] args) {
        Obj obj = new Obj(10,"안바껏지롱");
        System.out.println("처음 : "+obj);
        test(obj);
        System.out.println("끝 : "+ obj);

    }

    static void test(Obj prev) {
        prev.num = 100;
        prev.str = "바껏지롱ㅋㅋ";
        System.out.println("메소드 내부 : " + prev);
    }
}
```

```
<terminated> Just_Practice (1) [Java Application] C
처음 : Obj [num=10, str=안바껏지롱]
메소드 내부 : Obj [num=100, str=바껏지롱ㅋㅋ]
끝 : Obj [num=100, str=바껏지롱ㅋㅋ]
```

이렇게 바뀌는 이유는 참조타입의 변수는 Heap Memory영역에 생성된 객체의 주소값을 참조하기 때문이다.

따라서 위의 코드에서 객체의 값이 변하게 된 이유는 **메인메소드 내의 obj객체도 Heap메모리 영역에 있는 객체의 주소값을 참조**하고 있고 **test메소드에 넘겨준 인자값도 객체의 주소값**이기 때문에 결국 객체의 주소값을 통해 값을 변경하였으므로 변하게 된 것이다.

하지만 만약 test메소드 내에서 새로운 객체(new)를 할당하게 된다면?

```java
package Practice;

import java.util.*;

public class Just_Practice {
    static class Obj{
        int num;
        String str;

        public Obj(int num, String str) {
            super();
            this.num = num;
            this.str = str;
        }

        @Override
        public String toString() {
            return "Obj [num=" + num + ", str=
        }
    }

    public static void main(String[] args) {
        Obj obj = new Obj(10,"안바꼇지롱");
        System.out.println("처음 : "+obj);
        test(obj);
        System.out.println("끝 : "+ obj);

        String str = new String("으아아아아아");

    }

    static void test(Obj prev) {
        prev = new Obj(100000,"바꼇니?");
        System.out.println("메소드 내부 : " + prev);
    }
}
```

```
<terminated> Just_Practice (1) [Java Application]
처음 : Obj [num=10, str=안바꼇지롱]
메소드 내부 : Obj [num=100000, str=바꼇니?]
끝 : Obj [num=10, str=안바꼇지롱]
```

이와 같은 경우엔 test메소드에서 새로운 객체를 할당하였으므로 **Heap 영역에 새로운 객체가 생성**된다.

따라서 메인메소드에 있는 obj 객체는 값이 변화가 없으므로 변하지 않는다.

배열도 마찬가지로 자바의 참조변수이므로 같은 결과를 갖는다.

```java
package Practice;

import java.util.*;

public class Just_Practice {
    public static void main(String[] args) {
        int a[] = {0};
        test(a);
        System.out.println(a[0]);
    }

    static void test(int[] a) {
        a[0] = 10;
    }
}
```

```
<terminated> Ju
10
```

> 인자의 주소에 접근해 값을 변경하므로 변경된 값이 출력된다.

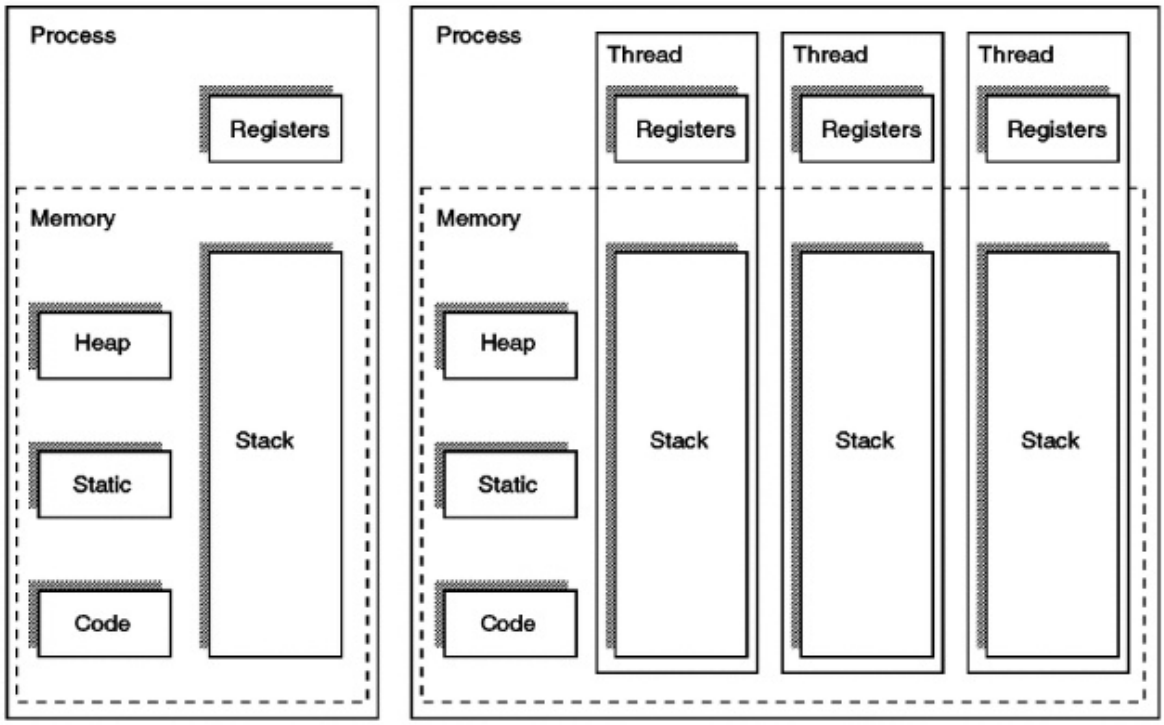
```java
package Practice;

import java.util.*;

public class Just_Practice {
    public static void main(String[] args) {
        int a[] = {0};
        test(a);
        System.out.println(a[0]);
    }

    static void test(int[] a) {
        a = new int[] {100};
    }
}
```

```
<terminated
0
```

> test메소드에서 새로운 참조변수를 생성하므로 기존 배열이 변경되지 않는다.

Thread Safe

데이터 공유

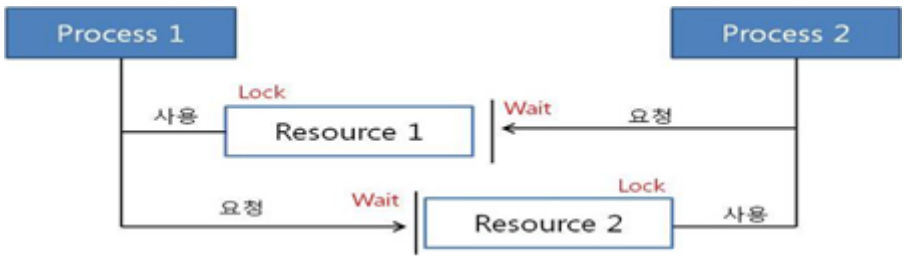


- Thread는 각각 별도의 스택을 가지긴 하나 프로세스에 비해 매우 작은 영역이다.
- 따라서 Process의 메모리 영역을 모든 Thread가 공유한다.
- 메모리의 접근 즉, 데이터의 접근은 그 무결성과 정합성이 지켜져야 한다.

> 메모리 공유에 의한 데이터 무결성과 정합성을 지키기 위해 안전한 쓰레드의 접근 Thread-Safe가 필요하게 되었다.

Thread-Safety

스레드 안전(Thread-Safety)이란 Multi-Thread 프로그래밍에서 일반적으로 어떤 함수나 변수, 혹은 객체가 여러 Thread로부터 동시에 접근이 이루어져도 프로그램의 실행에 문제가 없음을 뜻함. 보다 엄밀하게는 하나의 함수가 한 Thread로부터 호출되어 실행 중일 때, 다른 Thread가 그 함수를 호출하여 동시에 함께 실행되더라도 각 Thread에서 함수의 수행 결과가 올바로 나오는 것을 정의한다.

> Deadlock의 예시
>
> 
>

Thread-safe를 지키기 위한 방법

1. Re-entrancy : 어떤 메서드가 한 thread에 의해 호출되어 실행 중일 때, 다른 thread가 그 메서드를 호출하더라도 그 결과가 각각에게 올바로 주어져야 한다.
2. Thread-local storage : 공유 자원의 사용을 최대한 줄여 각각의 thread에서만 접근 가능한 저장소들을 사용함으로써 동시 접근을 막는다.

   이 방식은 동기화 방법과 관련되어 있고, 또한 공유상태를 피할 수 없을 때 사용하는 방식이다.
3. Mutual exclusion : 공유 자원을 꼭 사용해야 할 경우 해당 자원의 접근을 세마포어 등의 락으로 통제한다.
4. Atomic operations : 공유 자원에 접근할 때 원자 연산을 이용하거나 '원자적'으로 정의된 접근 방법을 사용함으로써 상호 배제를 구현할 수 있다.