JAVA

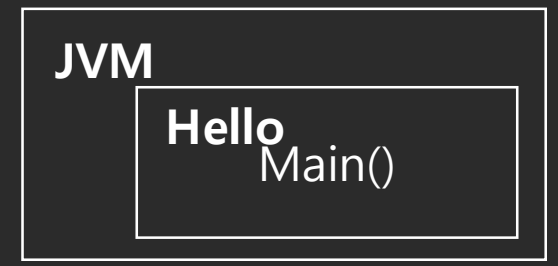
소스 : https://github.com/hyomee/JAVABASIC

1. JAVA

01. JAVA File 구조

- 자바 소스 파일의 확장자는 ,java
- 자바 파일명은 접근 지정자가 public인 Top Level Class가 있다면 Class Name 으로 되어야 함, 없다면 아무 이름으로 사용 할 수 있음
- Public Class 가 main Method를 가진다.
- package 구가 있다면, 해당 자바파일은 반드시 패키지명의 폴더에 존재해야 한다.

```java
package javahello;

public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World!!");
    }
}
```



02. Class를 객체화 하여 사용

- new()

03. Classpath

- Class를 찾는 경로
- 설정방법

: Java 실행시 –classpath option을 사용 하여 class를 찾는 경로 지정
: OS에서 환경설정 (set classpath=~ -> c:₩;
=> 현재 디렉토리 포함 하기 하려면 c:₩;. [dot] 사용)
: 지정 하지 않으면 현재 디렉토리

- Java Launcher의 Class 찾기

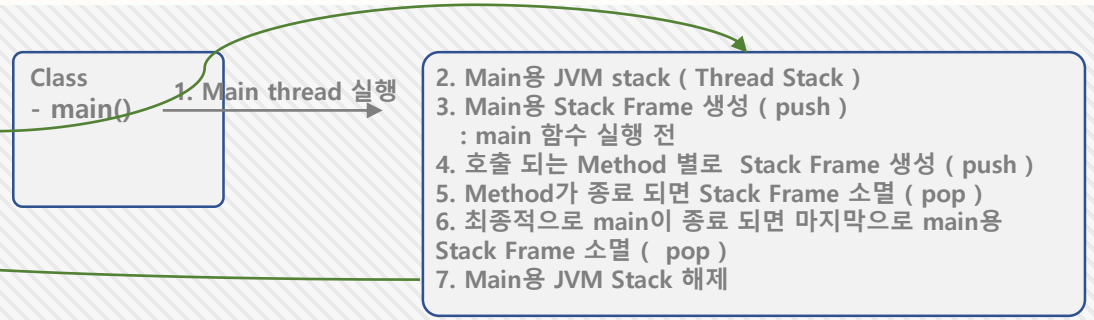
: 자바 플랫폼을 구성하는 클래스들이며 rt.jar(Object.class, String.class)에 포함되어 있는 클래스
: jre/lib/ext 확장 디렉토리에서 모든 jar 파일들을 자바2 확장 클래스
: 사용자가 환경 변수에서 지정한 경로에서 클래스

```java
package javahello;
class Exp {
    String expName;
    public String getExpName() {
        return expName;
    }
    public void setExpName(String expName) {
        this.expName = expName;
    }
}

public class Hello {
    public static void main(String[] args) {
        Exp exp = new Exp();
        exp.setExpName("홍길동");
        System.out.println(exp.getExpName() + " Hello World!!");
    }
}
```

2. JVM

01. JVM

- JRE(Java Runtime Enviroment)는 크게 API, JVM으로 구성 됨
- JVM(자바 가상 머신, Java Virtual Machine)은 클래스 로더를 통해 자바 클래스를 메모리로 로드하여 자바API를 이용하여 실행한다.
- **Method안에서 선언한 로컬 데이터는 Thread로 부터 안전 하다는 의미는** JVM Stack에 저장 된 데이터는 해당 Thread에서만 사용 할 수 있기 깨문 이다,
- 객체는 new연산자에 의해 메모리 heap에 생성 되고 JVM의 GC(Garbage Collector)에 의해 자동으로 Heap 메모리에서 해제 됨.



02. 오류

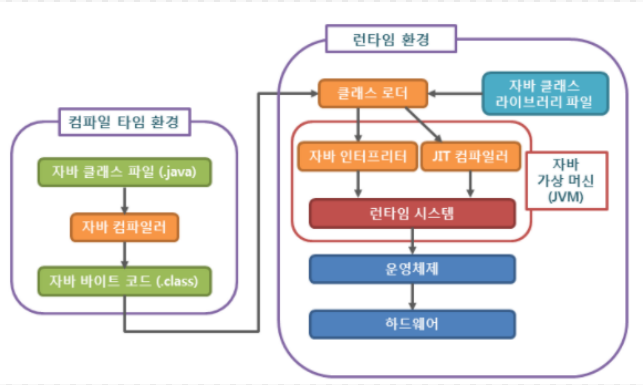
- StrackOverflowError
  : Stack Frame에 Method를 추가 할 공간이 없을 때 발생
  : JVM –Xss 옵션을 사용 하여 크기 조정
- OutOfMemoryError
  : 실행 중인 Thread가 많아서 JVM Stack를 할당 할 수 없을 때 발생

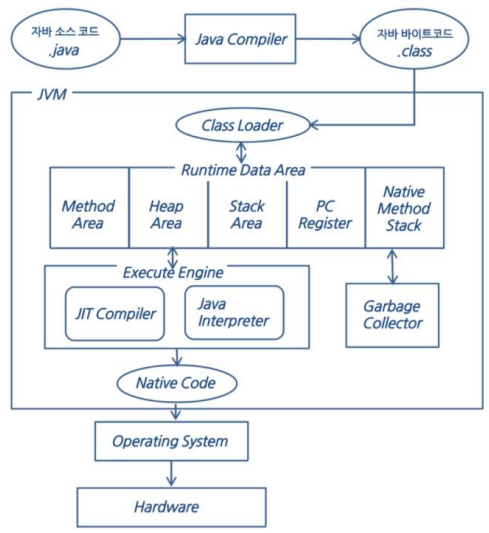
03. JVM Data Type

- 기본 자료형 4Byte -> 플랫폼 독립성 보장

04. 실행 과정

- 자바프로그램을 실행하면 JVM의 클래스 로더가 컴파일 된 자바 바이트코드(.class 파일)을 런타임 데이터 영역(Runtime Data Area)의 Method Area에 로드하고 실행 엔진(Execution Engine)이 이를 기계어로 변역 하면서 실행.



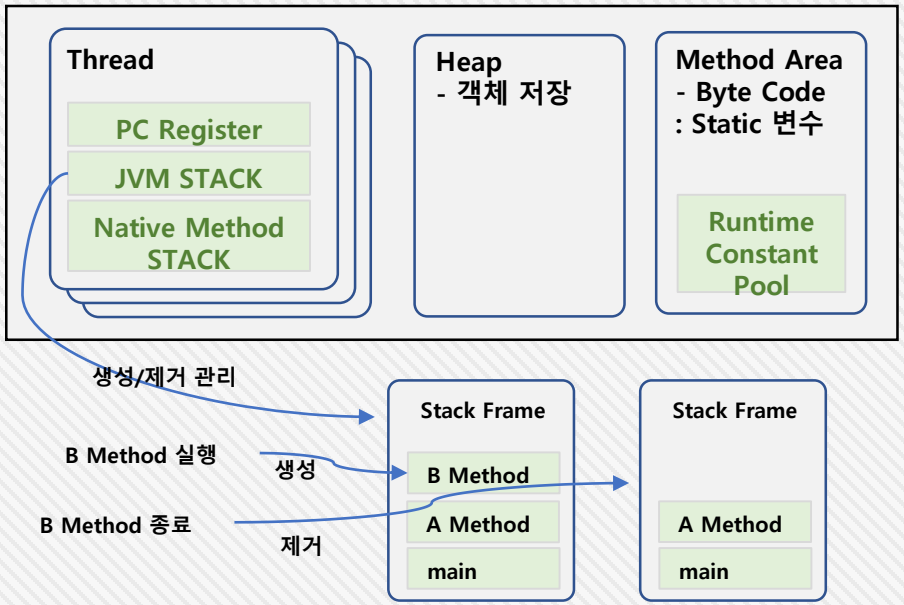


ABACUS

2. JVM

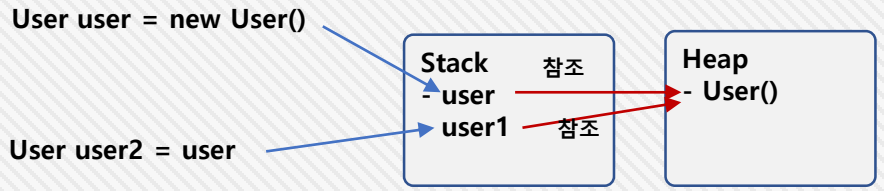
01. JVM Runtome Data Area

- 운영체계로 부터 부여 받은 메모리 영역
- **Method Area**
  : 모든 쓰레드가 공유 하는 영역, JVM이 시작 할 때 생성, 클래스와 인터페이스 Method에 대한 바이트 코드, 전역변수, 런타임 상수등이 저장됨-> **Main Method가 컴파일 된 Byte Code가 있음**
- Heap Area
  : 객체를 저장 할 때 사용 하는 영역 => 성능 고려 필요
- **JVM Stack (임시 메모리)**
  : 실행 시 Stack Frame이라는 각 쓰레드 마다 하나씩 할당
  : 실행되는 메소드의 **Stack Frame에는 지역변수, 메소드의 인자, 메소드의 리턴값, 리턴 번지 등이 저장되고** Stack Frame은 메소드가 끝나면 사라짐
- **Program Counter Register**
  : 쓰레드마다 하나씩 존재 : JVM의 명령어 주소
- **Runtime Constant Pool**
  : Method Area에 할당, 상수, 메소드, 필드를 저장
  : 자바 프로그램이 참조 할 경우 메모리 주소를 찾아서 참조함
- **Native Method Stack**
  : 자바 이외의 언어로 작성된 코드를 위한 Stack (C, C++ 등)



02. JVM Runtome Data Area

- 매소드 내에서 객체 참조 하면 선언한 변수는 지역변수로 Stack에 위치 하여 Heap에 저장 된 객체에 대한 참조값을 가짐
- New 연산자는 Heap 메모리에 객체를 만들고 그 객체의 참조 값을 반환 함



ABACUS

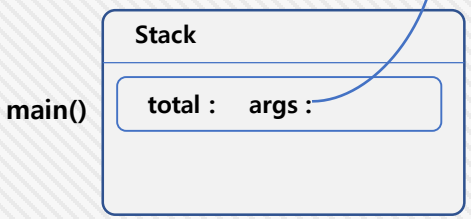
2. JVM

03. JYM 상태

```
class ClassMain {
    static int counter;

    public static void main(String[] args) {
        int total = sum(10, 30);
    }

    static int sum(int i, int j) {
        int sum = i + j;
        counter = counter + 1;
        return sum
    }
}
```

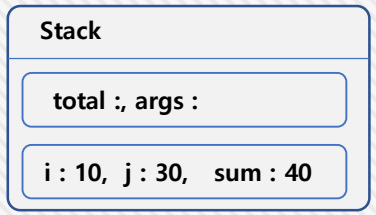
1. ClassMain Class가 시작 할 때 할당 됨

1. 메인 실행

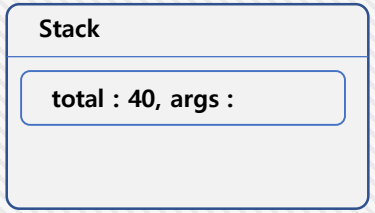


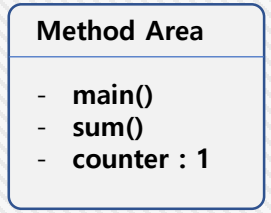
main()

2. SUM 실행



3. SUM 종료





* MAIN 종료 후 모두 사라짐

3. 변수

01. 지역 변수 (로컬변수)

- 메소드 내부에서 정의 되어 사용 하는 변수
- 자동으로 초기화 되지 않음
- 매개변수도 지역 변수
  : 메소드이 인자로 사용되는 변수

02. 인스턴스 변수

- static 으로 선언 되어 있지 않는 모든 멤버 변수
- 객체(클래스의 인스턴스)는 자신만의 복사본을 Heap에 저장 함
  : new로 생성시 마다 Heap에 할당
  : 인스턴스 변수의 값은 각각이 객체와 구분 됨

```
User userA = new User();
User userB = new User();

**************************************
userA,name = :홀길동
userB,name = 홈당무;
```

03. 클래스 변수

- 객체(클래스의 인스턴스)가 아니라 정의된 클래스와 연관되므로 Runtime Data Area의 Method Area에 한 개 존재
  : 객체를 많이 생성 해도 하나만 존재 함
  : 초기화가 한번만 실행
- : static 한정자
- 생성시점 : 최초 new하는 경우 , Class가 최초로 참조 되는 경우
- 일반적으로 상수로 사용
  : static final double PI=3.14;
- Class.클래스변수로 접근
  : ClassName.PI

04. 변수 자동 초기화

- 클래스, 인스턴스 변수는 자동 초기화 됨
  : boolean -> false
  : char -> '₩u0000'
  : Byte : short : int : long -> 0
  : Float -> 0.0f
  : Double -> 0.0d
  : Object type -> null
- 자동으로 초기화 되지 않음
- 매개변수도 지역 변수
  : 메소드이 인자로 사용되는 변수

```
public class AutoInitVariable {

    boolean aBoolean;     char aChar;
    Byte aByte;           short aShort;
    int anInt;            long aLong;
    Float aFloat;         Double aDouble;
    Object object;

    public AutoInitVariable(){
    }

    public void printVarible() {
        System.out.println(String.format("boolean aBoolean :: %s", aBoolean));
        System.out.println(String.format("char aChar :: %s", aChar));
        System.out.println(String.format("Byte aByte :: %s", aByte));
        System.out.println(String.format("short aShort :: %s", aShort));
        System.out.println(String.format("int anInt :: %s", anInt));
        System.out.println(String.format("long aLong :: %s", aLong));
        System.out.println(String.format("Float aFloat :: %s", aFloat));
        System.out.println(String.format("Double aDouble :: %s", aDouble));
        System.out.println(String.format("Object object :: %s", object));
    }
}
```

```
boolean aBoolean :: false
char aChar ::
Byte aByte :: null
short aShort :: 0
int anInt :: 0
long aLong :: 0
Float aFloat :: null
Double aDouble :: null
Object object :: null
```

4. 배열

01. 배열

- 같은 데이터 Type를 가지는 여러 값을 저장
- 선언 : 대괄호로 변수의 타입을 지정
  : 크기를 명시 하지 않음
  : 타입 -> 원시 데이터 (int, long), 참조 유형 (Class, 객체)
  : 예) int[] a;
- 생성

```
: int[]     a = {1, 2, 3};          // int형 배열 선언 및 값 할당
  int       a1[];                   // int형 배열 선언
  a1 = new int[] {1,2,3,4,5};       // 배열 생성
```

```
#### 크기 :: 3
a1 :: 3
a1 :: 2
a1 :: 1
a1 :: 4
a1 :: 6
a1 :: 5

#### 크기 as ::3
as :: array
as :: of
as :: string

#### 크기 as = as1 ::2
another
array

#### 정렬
a1 sort 오름 차순 :: another
a1 sort 오름 차순 :: array
a1 sort 내림 차순 :: array
a1 sort 내림 차순 :: another
```

```java
public void expArray04() {
    int[] a = {1, 2, 3};              // int형 배열 선언 및 값 할당
     int a1[];                        // int형 배열 선언
     a1 = new int[]{3, 2, 1, 4, 6, 5};          // 배열 생성

     System.out.println("#### 크기 :: " + a.length);
    for (int i : a1) System.out.print("a1 :: " + i + "\n");

    String[] as = {"array", "of", "string"};
    String[] as1 = {"another", "array"};


    System.out.println("\n#### 크기 as ::" + as.length);
    Arrays.asList(as).forEach(s -> System.out.println("as :: " + s));

    as = as1;
    System.out.println("\n#### 크기 as = as1 ::" + as.length);

    Arrays.asList(as).forEach(System.out::println);
    System.out.println("\n#### 정렬");
    // 정렬
     Arrays.sort(as);
    Arrays.asList(as).forEach(s -> System.out.println("a1 sort 오름 차순 :: " + s));

    Arrays.sort(as, Collections.reverseOrder());
    Arrays.asList(as).forEach(s -> System.out.println("a1 sort 내림 차순 :: " + s));
}
```

5. forEach

01. forEach

- 사용법 : collection.forEach(변수 -> 반복처리(변수))
- 문법 : java v1.8
  : @FunctionalInterface
  public interface Consumer<T> {
  void accept(T t)
  }
- void forEach(Consumer<T> action)
- 힘수형 인터페이스 : 추상 메소드기 히나인 인터페이스
  : accept : 인자로 받아서 리턴 하지 않음

```
#### Iterator
Iterator :: 값 = 파이선
Iterator :: 값 = 자바
#### for
for :: 값 = 파이선
for :: 값 = 자바
#### UserConsumer
UserConsumer :: 값 = 파이선
UserConsumer :: 값 = 자바
#### forEach
forEach :: 값 = 파이선
forEach :: 값 = 자바
#### System.out::println
파이선
자바
```

```java
class UserConsumer implements Consumer<String> {
    public void accept(String s) {
        System.out.println("Consumer impl :: " + s);
    }
}

public class ExpForEach {
    private List<String> list ;
    public ExpForEach() {
        String[] strArr = new String[]{"파이선", "자바"};
        list = (List) Arrays.asList(strArr);
    }

    public void printForEach() {

        System.out.println("#### Iterator ");
        Iterator<String> iter = list.iterator();
        while (iter.hasNext()) {
            System.out.println(String.format("Iterator :: 값 = %s", iter.next()));
        }

        System.out.println("#### for ");
        for( String str: list) {
            System.out.println(String.format("for :: 값 = %s", str));
        }

        System.out.println("#### UserConsumer ");
        list.forEach(new UserConsumer() {
            public void accept(String s) {
                System.out.println(String.format("UserConsumer :: 값 = %s", s));
            }
        });

        System.out.println("#### forEach ");
        list.forEach(str -> System.out.println(String.format("forEach :: 값 = %s", str)));

        System.out.println("#### System.out::println ");
        list.forEach(System.out::println);

    }
}
```

6. 클래스와 객체

01. 클래스

- 변수와 메소드를 정의 하는 프로토타입
- Field(맴버변수:객체의상태) + Operation(Method:객체의 행위)
- 클래스 이름은 대문자로 시작, 다음 단어의 시작은 대문자
- 사용자 정의 자료형, 객체의 자료형 (Sample **sample** = new Sample())
- Class 키워드로 선언, **논리적인 개체**, 한번만 선언
- 선언 시 키워드

  : **public** - 접근지정자가 맨 처음, : **abstract** - 추상클래스를 선언
  : **final** - 더 이상 자식으로 상속되지 않음을 명시, : **ClassName** - 클래스 이름
  : **extends** - 다른클래스를 상속, : **implements** - 인터페이스 구현)
- 초기화 순서

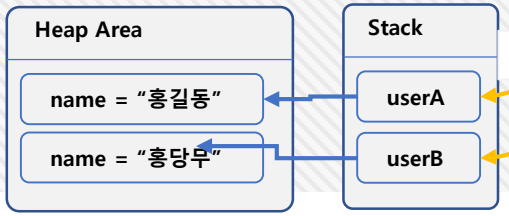
  : 메모리에 적재된 후 한 번 초기화
  - 모든 클래스 변수 (static 변수) 가 디폴트 값으로 초기화

02. 객체

- new 키워드에 의해서 만들어지며, 클래스의 인스턴스, **물리적인 개체**
  필요할 때 마다 생성
- Type이 Class인 변수
- 객체 이름은 소문자로 시작, 다음 단어의 시작은 대문자

```java
public class User {  // Class 정의
    private String name ;
    public User(String name) {
        this.name = name;
    }
    public void printNamePrint() {
        System.out.println(String.format("당신의 이름은 %s
    }
}
```

```java
public class UserMain {
    public static void main(String[] name) {
        User userA= new User("홍길동");  // 객체 생성
        userA.printNamePrint();

        User userB= new User("홍당무"); // 객체 생성
        userB.printNamePrint();

        System.out.println("#### Main 진입");
        new StaticInit();
    }
}
```




```java
class MemberVarTest {
    private int i = 0;
    public MemberVarTest(int i) {
        this.i = i;
        System.out.println(String.format("#### MemberVarTest : i: %s  ", i));
    }
}

public class StaticInit {
    private MemberVarTest memberVarTestA = new MemberVarTest(1);
    private MemberVarTest memberVarTestB;
    public static int i ;
    public static int num[] = new int[3];
    static {
        System.out.println("#### Class 초기화 블럭 실행");
        for (int i = 0 ; i < 3; i++) {
            num[i] = i;
        }
        System.out.println("#### Class 초기화 블럭 종료");
    }

    {
        memberVarTestB = new MemberVarTest(2);
    }

    public StaticInit() {
        System.out.println("### StaticInit 진입시작 ");
        System.out.println(String.format("#### StaticInit : static i: %s  ", i));
        System.out.println(String.format("#### StaticInit : num length : %s  ", num.length));
        for (int i = 0 ; i < num.length; i++) {
            System.out.println(String.format("#### StaticInit : static num[%s] : %s", i, num[i]));
        }
    }
}
```

```
#### Main 진입
#### Class 초기화 블럭 실행
#### Class 초기화 블럭 종료
#### MemberVarTest : i: 0
#### MemberVarTest : i: 2
### StaticInit 진입시작
#### StaticInit : static i: 0
#### StaticInit : num length : 3
#### StaticInit : static num[0] : 0
```

6. 클래스와 객체

03. 객체 생성자

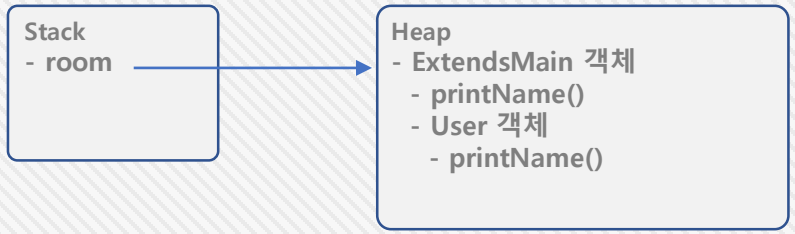
- new 연사자에 의해서 간접적으로 호출
  : 메모리 할당 , 생성자 호출, 객체 초기화(인스턴스 변수 초기화) or 인스턴스 블록 실행
  : 클래스이름과 같은 메소드 이름이 이며 리턴 타입이 없음
  : 클래스에 생성자는 없어도 됨 (기본 생성자)
  : 첫 문장에 있어야 함
- this
  : 자기 자신 객체 참조
  : 인스턴스 Method 내애서만 사용
  : 파라메터로 전달, 객체 참조 값 반환 가능

```
public class Constructor extends Object {
    String name ;  // 인스턴스 변수

    // 기본 생성자
    public Constructor() {
        // System.out.println("#### Constructor 기본 생성자 "); -> 주석을 풀면 오류 발생
        this("홀길동");   // 다른 생성자 호출
        this.name = "김길자";

        // 1. 다른 생성자 호출 ( 자기 자신의 또 다른 생성자 ) -> super(), this()
        // 2. super(), this() 를 이용한 다른 생성자 호출 시는 생성자의 첫 문자에 나타내야 한다.
        //    -> 컴파일 시점 오류 : java: call to super must be first statement in constructor
        //    -> super()을 사용해서 상위 클래스를 호출 하려면 this()에 의헤서 자기 자신 생성자를 호출
        //       하였을 경우는 다른 생성자에서 호출 하여야 한다.

        System.out.println(String.format("#### 기본 생성자 Constructor() :: %s", this.name));
    }

    // 파라메터 하나인 생성자
    public Constructor(String name) {
        super();
        this.name = name;
        System.out.println(String.format("#### 생성자 Constructor(String name) :: %s", this.name));
    }
}

결과 ================================================================================
#### 생성자 Constructor(String name) :: 홀길동
#### 기본 생성자 Constructor() :: 김길자
```

7. 상속

01. 상속

- 클래스를 확장하여 새로운 자식 클래스를 만드는 것
  : 부모 – 자식 관계
  : 부모 클래스의 기능(필드, 메소드)를 사용 (코드 재사용)
  : private로 선언된 것 아닌 것에만 접근 가능 함
  : extends 키워드를 사용 해서 오직 한 개만 상속이 가능 함
  : final 키워드로 상속을 막을 수 있음
  : protected로 선언된 Method는 상속된 객체만 사용
- 생성자
  : 생성자는 상속 되지 않음, 자식 클래스의 생정자에 의해 반드시 호출 (super(..))
  - 부모 클래스의 멤버를 초기화 할 수 있음
  : 자식 클래스가 생성 될 때 부모 상속 받은 변수를 저장할 수 있는 메모리를 포함하여 객체 할당




```
public class ExtendsMain {
    public static void main(String[] name) {
        Room room = new Room("홍길동",  "18", "7");
        room.printRoom();
        room.printName();
    }
}


class User {
    public String name;
    private String ages;

    public User(String ages) {
        this.ages = ages;
    }

    public void printName() {
        System.out.println(String.format("이름 : %s", this.name));
        System.out.println(String.format("나이 : %s", this.ages));
    }
}

class Room extends User {
    private String roomNum ;

    public Room(String name, String ages, String roomNum) {
        super(ages);
        this.name = name;
        this.roomNum = roomNum;
    }

    public void printRoom() {
        System.out.println(String.format("%s 은 %s번반에 있습니다", this.name, roomNum));
    }
}

결과 ============================================================
이름 : 홍길동
나이 : 18
홍길동 은 7번방에 있습니다
```

7. 상속

02. 추상

- 추상 클래스
  : 추상 Method를 하나 이상 포함하면 반드시 추상 클래스
  : abstract로 정의된 클래스로 추상 Method가 없어도 관계 없음
  : new 연산자로 생성 할 수 없음
- 추상 Method
  : Method 선언만 있고 구현이 없는 Method
  : abstract 리턴타입 Method명 () ;
  : 자식 클래스에서 구현 해야 함 (**Overriding**) – 각자 알아서 구현 (업무에 맞게,,,,)

다양성

- **Overriding**
  : 상위 클래스에 정의된 함수와 동일한 형태의 함수를 하위 클래스에 정의
- **Overloading**
  : 메소드의 매개변수의 유형과 개수가 다르게 하면서 동일한 메소드를 정의

```
PrintNm[] printNms = new PrintNm[3];
printNms[0] = new Mammalia();
printNms[1] = new Reptile();
printNms[2] = new Pisces();

for(PrintNm printNm : printNms) {
    printNm.printing();
}

결과 ==========================
포유류
파출류
어류
```

```
interface PrintNm {
    void printing();
}

class Mammalia implements PrintNm {
    @Override
    public void printing() {
        System.out.println(String.format("포유류"));
    }
}
class Reptile implements PrintNm {
    @Override
    public void printing() {
        System.out.println(String.format("파출류"));
    }
}
class Pisces implements PrintNm {
    @Override
    public void printing() {
        System.out.println(String.format("어류"));
    }
}
```

```
abstract class AbstractUser {
    public String name;
    private String ages;

    public AbstractUser(String ages) {
        this.ages = ages;
    }

    public void printName() {
        System.out.println(String.format("이름 : %s", this.name));
        System.out.println(String.format("나이 : %s", this.ages));
    }

    // 추상 메소드
     abstract void work();
}

class AbstractRoom extends AbstractUser {
    private String roomNum ;

    public AbstractRoom(String name, String ages, String roomNum) {
        super(ages);
        this.name = name;
        this.roomNum = roomNum;
    }

    // 추상 메소드 구현
     @Override
    void work() {
        System.out.println(String.format("추상 :: %s 은 %s번방에서 일하고 있습니다", this.name, roomNum));
    }

    public void printRoom() {
        System.out.println(String.format("%s 은 %s번방에 있습니다", this.name, roomNum));
    }
}

결과 ================================================================
이름 : 홍길동
나이 : 18
홍길동 은 7번방에 있습니다
추상 :: 홍길동 은 7번방에서 일하고 있습니다
```

8. 익명 클래스

01. Object Class

- 모든 클래스의 최상위 클래스
  : Class 만들 때 아무것도 상속을 받지 않으면 컴파일러에 의해서 자동으로 선언됨
  (extends Object)
  : Object 타입의 변수는 어떠한 객체도 가리킬 수 있음
  - Object obj = new int[5];
  - Object obj = new StringBuffer("abc");
  - Object obj = new Userclass();

02. 익명 클래스(annoymous class)

- 이름 없는 클래스
  : Method 안에 만들어짐
  : 클래스를 정의하지 않고 필요할 때 이름없이 즉시 선언하고 인스턴스화 해서 사용
  : 객체 안에 만드는 로컬 클래스와 동일 하다, (이름이 없는 것을 제외 하면)
  : 형식 :: new 클래스이름(or 인터페이스 이름) (...) {...}
  : new 수식이 올 수 있는 곳 어디든지 사용 가능하나 생성자는 정의 할 수 없음
  : 익명 클래스내부에서 외부의 메소드 내 변수를 참조할 때는 메소드의 지역 변수 중 final로 선언된 변수만 참조 가능
  - 변수는 Stack에 있고 객체는 Heap에 있음, 즉 Method 실행 이 끝나고 Stack는 사라지지만 Heap에 있는 Method는 사라지지 않기 때문
- 해당 클래스나 인터페이스를 정의하여 사용 할 때 여러 곳에서 사용되는 것이 아니라 단 한번만 정의해서 사용 하는 경우에 유용

```java
public class AnnoymousClassMain {

    interface AnnoymisClass {
        public void printClassType();
    }

    public void sayHello(String name) {
        // 로컬 클래스
        class LocalClass implements AnnoymisClass {
            public void printClassType() {
                System.out.println("Hello " + name);
            }
        }

        AnnoymisClass localClass = new LocalClass();
        localClass.printClassType();

        // 익명 클래스
        AnnoymisClass annoymisClass = new AnnoymisClass() {
            public void printClassType() {
                System.out.println("익명 ~ " + name);
            }
        };

        annoymisClass.printClassType();
    }


    public static void main(String... args) {
        AnnoymousClassMain myApp = new  AnnoymousClassMain();
        myApp.sayHello("클래스");

    }
}
```

```
================================================
Hello 클래스
익명 ~ 클래스
```

9. 람다

자바8이전에는 Method라는 함수 형태가 존재하지만 객체를 통해서만 접근이 가능하고, Method 그 자체를 변수로 사용하지는 못한다.
자바8에서는 함수를 변수처럼 사용할 수 있기 때문에, 파라미터로 다른 메소드의 인자로 전달할 수 있고, 리턴 값으로 함수를 받을 수도 있다.

01. 람다식

- 이름없는 익명 함수 구현에서 주로 사용하며 함수형 인터페이스의 인스턴스(구현 객체)를 표현
  : 함수형 인터페이스 (추상 메소드가 하나인 인터페이스)를 구현 객체를 람다식으로 표현

02. 문법 상세

1. 인터페이스

```
Interface Example {
    R apply(A arg);
}
```

2. 인스턴스 생성

```
Example exp = new Example() {
    @Override
    public R apply(A arg) {
        body
    }
};
```

3. 인자 목록과 함수 몸통을 제외 하고 모두 제거

```
Example exp = (arg) {
    body
};
```

4. 문법 적용

```
Example exp = (arg) - >{
    body
} ;
```

- **(arg1, arg2...) -> { body }** **// body에 표현식이 없거나 한개이상 올 수 있다.**
- **(params) -> expression**
- **(params) -> statement**
- **(params) -> { statements }**
- **(int a, int b) -> { return a + b; };**
  : (a, b) -> { return a+b } ; **// 타입 추론에 의한 타입 제거**
  : (a, b) -> a+b; **// 무엇인가를 반환 하거나 한 줄 표현식이 가능 하면 return 삭제**
- **() -> System.out.println("Hello ");** **// 파라미터없고 Hello 출력**
- **System.out::println;**

- () -> System.out.println("Hello "); // 파라미터없고 Hello 출력
- (String s) -> { System.out.println(s); } // String s입력매개변수로 받아 출력
- () -> 8514790 //파라미터없고 8514790가 리턴
- () -> { return 3.14 }; //파라미터없고 3.14리턴

```
public class LambdaMain {
    public static void main(String[] args) {

        PrintNm printNm = (name) -> {
            System.out.println(String.format("이름 : %s", name));
        };

        printNm.printName("홍길동");
        printNm.defaultPrintName("홍길자");

        ArrayList<String> citys = new ArrayList<String>();
        citys.add("SEOUL");
        citys.add("BUSAN");

        System.out.println("### Consumer ====================");
        citys.forEach( new Consumer<String>() {
            public void accept(String s) {
                System.out.println(s);
            }
        });

        System.out.println("### Consumer I/F 구현객체를 람다로 구현 ====================");
        citys.forEach(s -> System.out.println(s));
        System.out.println("### Consumer I/F 구현객체를 람다로 구현 축약 ====================");
        citys.forEach(System.out::println);
    }
}


interface PrintNm {
    void printName(String name);

    // default를 쓰면 인터페이스 내부에서도 코드가 포함된 메
    default void defaultPrintName(String name) {
        System.out.println(String.format("이름 : %s", name));
    }
}
```

```
이름 : 홍길동
이름 : 홍길자
### Consumer ====================
SEOUL
BUSAN
### Consumer I/F 구현객체를 람다로 구현
SEOUL
BUSAN
### Consumer I/F 구현객체를 람다로 구현 축약
SEOUL
BUSAN
```

9. 람다

자바8이전에는 Method라는 함수 형태가 존재하지만 객체를 통해서만 접근이 가능하고, Method 그 자체를 변수로 사용하지는 못한다.
자바8에서는 함수를 변수처럼 사용할 수 있기 때문에, 파라미터로 다른 메소드의 인자로 전달할 수 있고, 리턴 값으로 함수를 받을 수도 있다.

03. 함수형 인터페이스

- 추상Method가 하나뿐인 인터페이스 (Single Abstract Method : SAM)
- 여러 개의 Default Method가 있을 수 있다.
- @FunctionalInterface 어노테이션은 함수형 인터페이스임
- Runnable, ActionListener, Comparable은 함수형 인터페이스
  : 자바 8 이전 : 익명 클래스 이용
  : 자바 8 이후 : 람다식 이용

04. java.util.function 에서 제공 하는 함수형 인터페이스

- Predicate: 하나의 매개변수를 주는 boolean형을 반환
- Consumer: 하나의 매개변수를 주는 void 형 accept 메소드
- Function: T 유형의 인수를 취하고 R 유형의 결과를 반환하는 추상 메소드 apply
- Supplier: 메소드 인자는 없고 T 유형의 결과를 반환하는 추상 메소드 get
- UnaryOperator: 하나의 인자와 리턴타입을 가진다. T -> T
- BinaryOperator: 두 개의 인수, 동일한 타입의 결과를 반환하는 추상 메서드 apply

```java
@FunctionalInterface
interface Calculation {
    Integer apply(Integer x, Integer y);
}

class CalculationClass {

    // 인터페이스와 두개의 인자를 받아서 계산하는 Method
    static Integer calculate(Calculation calculation, Integer x, Integer y) {
        return  calculation.apply(x, y);
    }

    // 람다 함수 생성
     private Calculation addition = (x, y) -> x+y;
    private Calculation subtraction = (x, y) -> x-y ;

    public CalculationClass(Integer x, Integer y) {
        // 함수 사용
         System.out.println(String.format("%s + %s = %s", x, y, calculate(addition, 2,2)));
         System.out.println(String.format("%s + %s = %s", x, y, calculate(subtraction, 2,2)));
```

```java
@FunctionalInterface
interface Worker {
    public void work();
}

class FunctionInterfaceTest {

    void execute(Worker worker) {
        worker.work();
    }

    public void runWorker() {
        execute(new Worker() {
            @Override
            public void work() {
                System.out.println("Worker 실행");
            }
        });

        execute( () -> System.out.println("Worker 람다식 실행"));
    }

}
```

```
== 결과 ============
Worker 실행
Worker 람다식 실행
```

9. 람다

자바8이전에는 Method라는 함수 형태가 존재하지만 객체를 통해서만 접근이 가능하고, Method 그 자체를 변수로 사용하지는 못한다.
자바8에서는 함수를 변수처럼 사용할 수 있기 때문에, 파라미터로 다른 메소드의 인자로 전달할 수 있고, 리턴 값으로 함수를 받을 수도 있다.

04. 타입추론

- 자바는 타입 추론을 지원 하지 않았지만 1.8이후 Method 호출 시 매개변수 타입 추론을 지원

```
@FunctionalInterface
interface Calculation {
    Integer apply(Integer x, Integer y);
}

class CalculationClass {

    // 인터페이스와 두개의 인자를 받아서 계산하는 Method
    static Integer calculate(Calculation calculation, Integer x, Integer y) {
        return  calculation.apply(x, y);
    }

    // 람다 함수 셍성 :: 인터페이스에 타입이 지정 되어 있음
    private Calculation addition = (x, y) -> x+y;
    private Calculation subtraction = (x, y) -> x-y ;

    public CalculationClass(Integer x, Integer y) {
        // 함수 사용
        System.out.println(String.format("%s + %s = %s", x, y, calculate(addition, 2,2)));
        System.out.println(String.format("%s + %s = %s", x, y, calculate(subtraction, 2,2) ));
    }

}
```

```
5 + 3 = 4
5 + 3 = 0
```

10. 제네릭

클래스 내부에서 사용하는 데이터의 타입(Type)을 클래스의 인스턴스를 생성할 때 결정하는 것을 의미.
객체의 타입을 컴파일 시점에 체크하기 때문에 타입 안정성을 높이고 형 변환의 번거로움을 줄일 수 있음.

01. 제네릭

- 제네릭(Generic)은 클래스 내부에서 사용하는 데이터의 타입(Type)을 클래스의 인스턴스를 생성할 때 결정하는 것.
- 객체의 타입을 컴파일 시점에 체크하기 때문에 타입 안정성을 높이고 형 변환의 번거로움을 줄일 수 있음.
- 기본 데이터 타입(int, long..)에 대해서는 지정이 불가능
- 사용법
  : public class 클래스명<T> {...}
  : public interface 인터페이스명<T> {...}
- 자주 사용 하는 타입인자

<T>	Type
<E>	Element
<K>	Key
<N>	Number
<V>	Value
<R>	Result

```
GenericClassSingle<String> genericString = new GenericClassSingle<String>();
genericString.memberVar = "문자";
genericString.printType();

GenericClassSingle<Integer> genericInteger = new GenericClassSingle<Integer>();
genericInteger.memberVar = 1;
genericInteger.printType();
```

```
class GenericClassSingle<T> {
    // example은 T type으로 인스턴스가 생성 될 때 결정이 된다.
    public T memberVar;

    public void printType () {
        System.out.println(String.format("Type :: %s, Value :: %s",
                                         memberVar.getClass().getTypeName(), memberVar));
    }
}
```

```
// 복수 제네릭
interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
```

복수 제네릭

```
class OrderedPair<K,V> implements Pair<K,V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    @Override
    public K getKey() {
        return key;
    }

    @Override
    public V getValue() {
        return value;
    }
}
```

```
// 복수 제네릭
// - 시용 시점에 형 결정 됨
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");

System.out.println(String.format(" p2 [ Key :: %s (Type : %s), Value :: %s (Type : %s)",
        p1.getKey(),
        p1.getKey().getClass().getTypeName(),
        p1.getValue(),
        p1.getValue().getClass().getTypeName()));

System.out.println(String.format(" p2 [ Key :: %s (Type : %s), Value :: %s (Type : %s)",
        p1.getKey(),
        p1.getKey().getClass().getTypeName(),
        p1.getValue(),
        p1.getValue().getClass().getTypeName()));
=====================================================================
 p2 [ Key :: Even (Type : java.lang.String), Value :: 8 (Type : java.lang.Integer)
 p2 [ Key :: Even (Type : java.lang.String), Value :: 8 (Type : java.lang.Integer
```

10. 제네릭

클래스의 메서드에서도 제네릭 메서드를 정의할 수 있으며 타입 매개변수의 사용은 메소드 내부로 제한 됨.

02. 제네릭 메서드

- 제네릭 메소드를 호출할 때는 실제 타입을 <> 안에 넣어줘도 되고 생략을 해도 됨
- 자료형을 매개변수로 가지는 메소드
- 하나의 메소드 정의로 여러 유형의 데이터를 처리할 때 유용함
- 메소드 정의에서 반환형 왼편, 각 괄호 <> 안에 타입 매개변수를 가짐
  : 타입 매개변수를 메소드의 반환형이나 메소드 인자의 타입으로 사용할 수 있음
  : 지역 변수의 타입으로 사용할 수도 있음

```
public static <T> T getLast(T[] a){
        return a[a.length-1];
}
```

- 인스턴스 메소드와 static 메소드 모두 제네릭 메소드로 정의 가능
- 제네릭 메소드를 호출할 때, 타입을 명시하지 않아도 인자에 의해 추론이 가능함

```
// 제너릭 메서드
class GenericMethod {

    public static <T> T printData(T data) {
        if(data instanceof String)
            System.out.println("String");
        else if(data instanceof Integer)
            System.out.println("Integer");
        else if(data instanceof Double)
            System.out.println("Double");

        return data;
    }

}
```

```
Integer aInt = GenericMethod.printData(1);
Double aDouble = GenericMethod.printData(1.0);
String aString = GenericMethod.printData("Stirng");
String aStringA =  GenericMethod.<String>printData("문자")


결과 ================================
Integer
Double
String
String
```

03. 제네릭 타입 제한

- 자료형을 매개변수화 하여 클래스/인터페이스/메소드를 정의할 때, 자료형에 제한을 두는 것
  : <T extends Number>와 같이 하면 T를 상한으로 정할 수 있음
  - 타입 매개변수는 Number의 서브 클래스라야 함

```
class GenericTypeBounded<T extends Number> {
    public void set(T value) {}
}

=========================================================
GenericTypeBounded<Integer> box = new GenericTypeBounded<>();
GenericTypeBounded.set("Hi"); // compile error
```

10. 제네릭

04. 제네릭 와일드 카드

- 와일드카드 타입에는 총 세가지의 형태가 있으며 물음표(?)라는 키워드로 표현
  : 제네릭타입<?>
  - 타입 파라미터를 대치하는 것으로 모든 클래스나 인터페이스타입이 올 수 있음
  : 제네릭타입<? extends 상위타입>
  - 와일드카드의 범위를 특정 객체의 하위 클래스만 올 수 있음.
  : 제네릭타입<? super 하위타입> :
  - 와일드카드의 범위를 특정 객체의 상위 클래스만 올 수 있음.

```
class Calcu {
    public void printList(List<?> list) {
        for (Object obj : list) {
            System.out.println(obj + " ");
        }
    }

    public int sum(List<? extends Number> list) {
        int sum = 0;
        for (Number i : list) {
            sum += i.doubleValue();
        }
        return sum;
    }

    public List<? super Integer> addList(List<? super Integer> list) {
        for (int i = 1; i < 5; i++) {
            list.add(i);
        }
        return list;
    }
}
```

04. 주의사항

- 기본 유형으로 제네릭 유형을 인스턴스화 할 수 없음
- 유형 매개 변수의 인스턴스를 생성 할 수 없음
- 유형이 유형 매개 변수 인 정적 필드를 선언 할 수 없음
- 매개 변수가있는 유형에 캐스트 또는 instanceof를 사용할 수 없음
- 매개 변수가있는 유형의 배열을 만들 수 없음
- 매개 변수가있는 유형의 개체를 생성, 캐치 또는 던질 수 없음
- 각 오버로드의 형식 매개 변수 유형이 동일한 원시 유형으로 지워지는 메서드를 오버로드 할 수 없음

11. 어노테이션(Annotation)

01. 어노테이션

- 자바 소스 코드에 추가하여 사용할 수 있는 메타데이터의 일종
- @기호를 앞에 붙여서 사용
- 자바 어노테이션은 클래스 파일에 임베디드되어 컴파일러에 의해 생성된 후 자바 가상머신에 포함되어 작동
- 메타데이터란 어플리케이션이 처리해야 할 데이터가 아니라 ,컴파일 과정과 실행 과정에서 코드를 어떻게 컴파일하고 처리할것인지를 알려주는 정보
- 어노테이션의 사용처
  1. 컴파일러에게 코드 문법 에러를 체크하도록 정보를 제공
  2. 소프트웨어 개발 툴이 빌드나 배치 시 코드를 자동으로 생성할 수 있도록 정보를 제공
  3. 실행 시 특정 기능을 실행하도록 정보를 제공
- 어노테이션의 필드에서는 enum, String이나 기본 자료형, 기본 자료형의 배열을 사용

02. 기본 제공 어노테이션

- @Override
  : 선언한 메서드가 오버라이드 되었다는 것.
  : 만약 상위(부모) 클래스(또는 인터페이스)에서 해당 메서드를 찾을 수 없다면 컴파일 에러를 발생
- @Deprecated
  : 해당 메서드가 더 이상 사용되지 않음을 표시, 만약 사용할 경우 컴파일 경고를 발생.
- @SuppressWarnings
  : 선언한 곳의 컴파일 경고를 무시.
- @SafeVarargs
  : Java7 부터 지원하며, 제너릭 같은 가변인자의 매개변수를 사용할 때의 경고.
- @FunctionalInterface
  : Java8 부터 지원하며, 함수형 인터페이스를 지정하는 어노테이션. 만약 메서드가 존재하지 않거나, 1개 이상의 메서드(default 메서드 제외)가 존재할 경우 컴파일 오류를 발생.

03. 기본 구조

```
@Target(ElementType.METHOD)                    // 메타 어노테이션
@Retention(RetentionPolicy.RUNTIME)            //  메타 어노테이션
public @interface CustomAnnotation {
   boolean isCheck() default true;
}
```

04. 메타 어노테이션의 종류

- @Retention : 자바 컴파일러가 어노테이션을 다루는 방법을 기술하며, 특정 시점까지 영향을 미치는지를 결정
  : RetentionPolicy.SOURCE : 컴파일 전까지만 유효. (컴파일 이후에는 사라짐)
  : RetentionPolicy.CLASS : 컴파일러가 클래스를 참조할 때까지 유효.
  : RetentionPolicy.RUNTIME : 컴파일 이후에도 JVM에 의해 계속 참조가 가능. (리플렉션 사용)
- @Target : 어노테이션이 적용할 위치를 선택 .
  : ElementType.PACKAGE : 패키지 선언
  : ElementType.TYPE : 타입 선언
  : ElementType.ANNOTATION_TYPE : 어노테이션 타입 선언
  : ElementType.CONSTRUCTOR : 생성자 선언
  : ElementType.FIELD : 멤버 변수 선언
  : ElementType.LOCAL_VARIABLE : 지역 변수 선언
  : ElementType.METHOD : 메서드 선언
  : ElementType.PARAMETER : 전달인자 선언
  : ElementType.TYPE_PARAMETER : 전달인자 타입 선언
  : ElementType.TYPE_USE : 타입 선언
- @Documented : 해당 어노테이션을 Javadoc에 포함.
- @Inherited : 어노테이션의 상속을 가능.
- @Repeatable : Java8 부터 지원하며, 연속적으로 어노테이션을 선언할 수 있게 해줌.

참고자료

제네릭 : https://docs.oracle.com/javase/tutorial/java/generics/index.html
https://docs.oracle.com/javase/tutorial/extra/generics/index.html

Annotation : https://elfinlas.github.io/2017/12/14/java-annotation/,
https://elfinlas.github.io/2017/12/14/java-custom-anotation-01/

THANKS

ABACUS

www.iabacus.co.kr

Tel. 82-2-2109-5400

Fax. 82-2-6442-5409