java.lang패키지는 자바프로그램에 가장 기본이 되는 클래스들을 포함하고 있으며, import문 없이도 사용할 수 있게 되어 있다.

● Object클래스

▶ equals(Object obj): 매개변수로 객체의 참조변수를 받아서 주소값으로 비교하여 그 결과를 boolean값으로 알려준다.

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

두 객채의 같고 다름을 참조변수의 값으로 판단한다. 그렇기 때문에 서로 다든 두 객체를 equals메서드로 비교하면 항상 false로 결과를 얻게 된다. 객체를 생성할 때, 메모리의 비어있는 공간을 찾아 생성하므로 서로 다든 두 개의 객체가 같은 주소를 갖는 일은 없기 때문이다. 두 개 이상의 참조변수가 같은 주소값을 같는 것(한 객체를 참조하는 것)은 가능하다. Object클래스로부터 상속받은 equals메서드는 두 개의 참조변수가 같은 객체를 참조하고 있는지, 즉, 두 참조변수에 저장된 값(주소값)이 같은지를 판단한다.

만약 주소가 아닌 객체에 저장된 내용을 비교하도록 하고 싶으면 equals메서드를 오버라이딩하면 된다.

```java
// 서로 다른 인스턴스일지라도 같은 id(주민등록번호)를 갖고 있다면 equals메서드로 비교했을 때
// true를 결과로 얻도록 하기

class Person {
    long id;

    public boolean equals(Object obj) {
        if (obj ! = null && obj instanceof Person) {
            return id == ((Person)obj).id;
            // obj가 Object타입이므로 id값을 참조하기 위해서는 Person타입으로 형변환 필요
        } else {
            return false; // 타입이 Person이 아니면 값을 비교할 필요도 없다.
        }
    }

    Person(long id) {
        this.id = id;
    }
}

class EqualsEx2 {
    public static void main(String[] args) {
        Person p1 = new Person(8011081111222L);
        Person p2 = new Person(8011081111222L);

        if (p1 == p2)
            System.out.println("p1과 p2는 같은 사람입니다.");
        else
            System.out.println("p1과 p2는 다른 사람입니다."); // 주소값 비교

        if(p1.equals(p2))
            System.out.println("p1과 p2는 같은 사람입니다."); // id 비교
        else
            System.out.println("p1과 p2는 다른 사람입니다.");

    }
}
```

String클래스 역시 Object클래스의 equals메서드를 오버라이딩해서 String인스턴스가 갖는 문자열 값을 비교하도록 되어 있다. 그렇기 때문에 같은 내용의 문자열을 갖는 두 String인스턴스에 equals메서드를 사용하면 항상 true값을 얻는 것이다. Data, File, wrapper클래스(Integer, Double 등)의 equals메서드도 주소값이 아닌 내용을 비교하도록 오 버라이딩되어 있다. 그러니 StringBuffer클래스는 오버라이딩되어 있지 않다.

▶ hashCode(): 해싱(hashing)기법에 사용되는 해시함수(hash function)을 구현한 메서드이다. 해싱은 데이터관리기법 중의 하나로 다량의 데이터를 저장하고 검색하는데 유용하다. 해시함수는 찾고자 하는 값을 입력하면, 그 값이 저장된 위치를 알려주는 해시코드(hash code)를 반환한다. hashCode메서드는 객체의 주소값으로 해시코드를 만들어 반환하기 때문에 32bit JVM에서는 서로 다든 두 객체는 결코 같은 해시코드를 가질 수 없지만 64 bit JVM에서는 8 byte 주소값으로 해시코드(4 byte)를 만들기 때문에 해시코드가 중복될 수 있다. 클래스의 인스턴스변수값으로 객체의 같고 다름을 판단해야 하는 경우라면 equals메서드 뿐 만아니라 hashCode메서드도 적절히 오버라이딩해야 한다. 같은 객체라면 hashCode메서드를 호출했을 때의 결과값인 해시코드도 같아야 하기 때문이다.

String클래스는 문자열의 내용이 같으면, 동일한 해시코드를 반환하도록 hashCode메서드가 오버라이딩되어 있기 때문에, 문자열의 내용이 같은 두 String인스턴스에 대해 hashCode()를 호출하면 항상 동일한 해시코드 값을 얻는다. 반면에 System.identityHashCode(Object x)는 Object클래스의 hashCode메서드처럼 객체의 주소값으로

해시코드를 생성하기 때문에 모든 객체에 대해 항상 다든 해시코드값을 반환할 것을 보장한다. 그래서 문자열 내용이 같은 두 String인스턴스의 해시코드는 같지만 서로 다른 객체라는 것을 알 수 있다.

```
class HashCodeEx1 {
    public static void main(String[] args) {
        String str1 = new String("abc");
        String str2 = new String("abc");

        System.out.println(str1.equals(str2));
        System.out.println(str1.hashCode());
        System.out.println(str2.hashCode());
        System.out.println(System.identityHashCode(str1));
        System.out.println(System.identityHashCode(str2));

        System.out.println("----------------------");
        String name = "홍길동";
        String name2 = "홍길동";

        if (name == name2) {
            System.out.println("같다");
        }

        if (name.equals(name2)) {
            System.out.println("같다");
        }

        if (System.identityHashCode(name) == System.identityHashCode(name2)) {
            System.out.println("같다");
        }

        System.out.println("----------------------");
        String name3 = new String("홍길동");
        String name4 = new String("홍길동");

        if (name3 != name4) {
            System.out.println("다르다");
        }

        if (name3.equals(name4)) {
            System.out.println("같다");
        }

        if (name3.hashCode() == name4.hashCode()) {
            System.out.println("같다");
        }

        if (System.identityHashCode(name3) != System.identityHashCode(name4)) {
            System.out.println("다르다");
        }
    }
}
```

▶ toString(): 인스턴스에 대한 정보를 문자열(String)로 제공할 목적으로 정의된 메서드.

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Object클래스에 정의된 toString()을 호출하면 클래스 이름에 16진수의 해시코드를 얻게 된다. 하지만 Stirng클래스와 Date클래스의 toString()은 각각 String인스턴스가 갖고 있는 문자열과 Date인스턴스가 갖고 있는 날짜와 시간을 문자열로 변환하여 반환하도록 오버라이딩되어 있다. 일반적으로 인스턴스나 클래스에 대한 정보 또는 인스턴스 변수들의 값을 문자열로 반환하여 변환하도록 오버라이딩 해서 사용하는 것이 보통이다. 오버라이딩 할 때, Object클래스에 정의된 toString()의 접근 제어자가 public이므로 오버라이딩 하는 클래스의 toString()의 접근제어자도 public으로 해야 한다(조상에 정의된 접근범위보다 같거나 더 넓어야 한다).

```
class Card {
    String kind;
    int number;

    Card() {
        this("SPADE", 1);  // Card(String kind, int number)를 호출
    }

    Card(String kind, int number) {
        this.kind = kind;
        this.number = number;
    }

    public String toString() {
        // Card인스턴스의 kind와 number를 문자열로 반환한다.
        return "kind : " + kind + ", number : " + number;
    }
}

class CardToString2 {
    public static void main(String[] args) {
        Card c1 = new Card();
        Card c2 = new Card("HEART", 10);
        System.out.println(c1.toString());
        System.out.println(c2.toString());
    }
}
```

▶ clone(): 자신을 복제하여 새로운 인스턴스를 생성하는 메서드. 원래의 인스턴스는 보존하고 clone메서드를 이용해서 새롱누 인스턴스를 생성하여 작업을 하면, 작업 이전의 값이 보존되므로 작업에 실패해서 원래의 상태로 되돌리거나 변경ㅌ되기 전의 값을 참고하는데 도움이 된다. Object클래스에 정의된 clone()은 단순히 인스턴스변수의 값만 복사하기 때문에, 참조타입의 인스턴스 변수가 있는 클래스는 완전한 인스턴스 복제가 이루어지지 않는다. 예를 들면 배열의 경우, 복제된 인스턴스도 같은 배열의 주소를 갖기 때문에 복제된 인스턴스의 작업이 원래의 인스턴스에 영향을 미치게 된다. 이런 경우, clone()메서드를 오버라이딩 해서 새로운 배열을 생성하고, 배열의 내용을 복사하도록 해야 한다.

```
public class Object {
    ...
    protected native Object clone() throws CloneNotSupportedException;
    ...
}
```

clone()을 사용하려면 먼저 복제할 클래스가 Cloneable인터페이스를 구현해야 하고, clone()을 오버라이딩 하면서 접근제어자를 protected에서 public으로 변경해야 한다. 그래야만 상속관계가 없는 다른 클래스에서 clone()을 호출할 수 있다. 또한 clone()은 반드시 CloneNotSupportedException 예외에 대한 예외처리를 해 주어야 한다. Cloneable인터페이스를 구현한 클래스의 인스턴스만 clone()을 통해서 복제가 가능한 이유는 인스턴스의 데이터를 보호하기 위해서이다. Cloneable인터페이스가 구현되어 있다는 것은 클래스 작성자가 복제를 허용한다는 의미이다.

```
class Point implements Cloneable {  // Cloneable인터페이스를 구현한다.
    int x;
    int y;

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

    public String toString() {
        return "x="+x +", y="+y;
    }

    public Object clone() {  // 접근제어자를 protected에서 public으로 변경한다.
        Object obj = null;
        try {
            obj = super.clone();  // try-catch내에서 조상클래스의 clone()을 호출한다.
           // clone()은 반드시 예외처리를 해주어야 한다.
        } catch(CloneNotSupportedException e) {}
        return obj;
    }
}

class CloneEx1 {
    public static void main(String[] args){
        Point original = new Point(3, 5);
        Point copy = (Point)original.clone(); // 복제(clone)해서 새로운 객체를 생성
        System.out.println(original);
        System.out.println(copy);
    }
}
```

☞ 공변 변환타입(covariant return type): 오버라이딩 할 때 조상 메서드의 반환 타입을 자손 클래스의 타입으로 변경을 허용하는 것. return문에 자손 클래스의 타입으로 형변환도 추가한다. 공변 반환타입을 사용하면, 조상의 타입이 아닌 실제로 반환되는 자손의 객체의 타입으로 반환할 수 있어서 번거로운 형변환이 줄어든다.

```
class Point implements Cloneable {  // Cloneable인터페이스를 구현한다.
    int x;
    int y;

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

    public String toString() {
        return "x="+x +", y="+y;
    }

    public Point clone() {  // 공변 변환타입: 반환타입을 Object에서 Point로 변경
        Object obj = null;
        try {
            obj = super.clone();
        } catch(CloneNotSupportedException e) {}
        return (Point)obj;  // Point타입으로 형변환
    }
}

class CloneEx1 {
    public static void main(String[] args){
        Point original = new Point(3, 5);
        Point copy = original.clone(); // 복제(clone)해서 새로운 객체를 생성
        System.out.println(original);
        System.out.println(copy);
    }
}
```

배열도 객체이기 때문에 Object클래스를 상속받으며, 동시에 Cloneable인터페이스와 Serializable인터페이스가 구현되어 있다. 그래서 Object클래스의 멤버들을 모두 상속받는다. clone()을 배열에서는 public으로 오버라이딩하였기 때문에 직접 호출이 가능하며, 원본과 같은 타입을 반환하므로 형변환이 필요 없다. 일반적으로는 배열을 복사할 때 같은 길이의 새로운 배열을 생성한 다음에 System.arraycoy()를 이용해서 내용을 복사하지만, clone()을 이용해서 간단하게 복사할 수도 있다.

```
import java.util.*;

class CloneEx2 {
    public static void main(String[] args){
        int[] arr = {1,2,3,4,5};

        // 배열 arr을 복제해서 같은 내용의 새로운 배열을 만든다.
        int[] arrClone = arr.clone();
        arrClone[0]= 6;
//      // 같은 길이의 새로운 배열을 생성하고 내용을 복사하는 것과 같다.
//      int[] arrClone = new int[arr.length];
//      System.arraycopy(arr, 0, arrClone, 0, arr.length);

        System.out.println(Arrays.toString(arr));         // [1, 2, 3, 4, 5]
        System.out.println(Arrays.toString(arrClone));    // [6, 2, 3, 4, 5]
    }
}
```

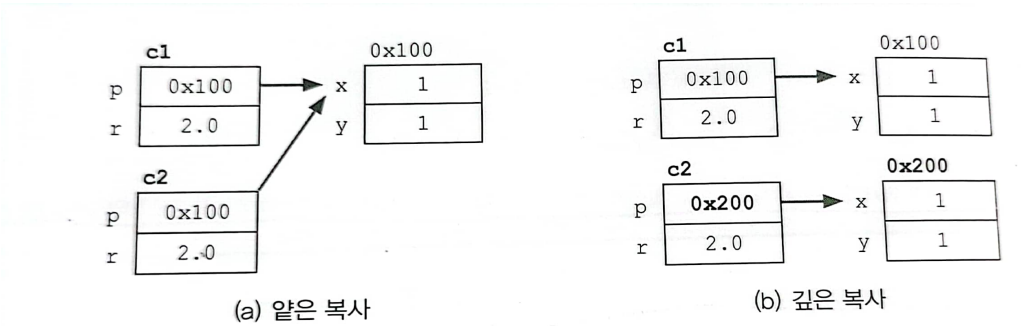
배열 뿐만 아니라 java.util패키지의 Vector, ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, Calendar, Date와 같은 클래스들이 이와 같은 방식으로 복제가 가능하다. clone()으로 복제가 가능한 클래스인지 확인하려면 Java API에서 Cloneable을 구현하였는지 확인하면 된다.

```
ArrayList list = new ArrayList();
ArrayList list2 = (ArrayList)list.clone();
```

☞ 얕은 복사와 깊은 복사: clone()은 단순히 객체에 저장된 값을 그대로 복제할 뿐, 객체가 참조하고 있는 객체까지는 복제하지 않는다. 때문에 객체배열을 clone()으로 복제하는 경우에는 원본과 복제본이 같은 객체를 공유하므로 완전한 복제가 아니고, 이러한 것을 얕은 복사(shallow copy)라고 한다. 얕은 복사에서는 원본을 변경하면 복사본도 영향을 받는다. Object클래스의 clone()은 원본 객체가 가지고 있는 값만 그대로 복사하므로 얕은 복사를 한다. 반면에 clone()을 오버라이딩 할 때 코드를 추가하면 원본이 참조하고 있는 객체까지 복제하게 할 수 있으며, 이를 깊은 복사(deep copy)라고 한다. 깊은 복사에서는 원본과 복사본이 서로 다른 객체를 참조하기 때문에 원본의 변경이 복사본에 영향을 미치지 않는다.

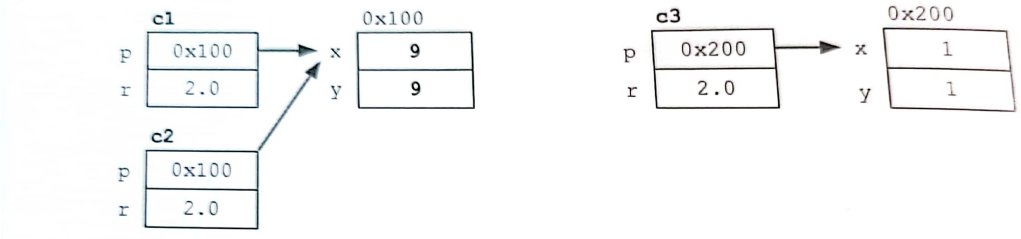
```
import java.util.*;

class Circle implements Cloneable {
    Point p;  // 원점. Point타입의 참조변수를 포함하고 있다.
    double r; // 반지름

    Circle(Point p, double r) {
        this.p = p;
        this.r = r;
    }

    public Circle shallowCopy() { // 얕은 복사
        Object obj = null;

        try {
            obj = super.clone();
        } catch (CloneNotSupportedException e) {}

        return (Circle)obj;
    }

    public Circle deepCopy() { // 깊은 복사
        Object obj = null;

        try {
            obj = super.clone();
        } catch (CloneNotSupportedException e) {}

        Circle c = (Circle)obj;
        c.p = new Point(this.p.x, this.p.y); // 복제된 객체가 새로운 Point인스턴스를 참조하
        // 원본이 참조하고 있는 객체까지 복사하므로 깊은 복사이다.

        return c;
    }

    public String toString() {
        return "[p=" + p + ", r="+ r +"]";
    }
}
```

```java
class Point {
    int x;
    int y;

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

    public String toString() {
        return "("+x +", "+y+")";
    }
}

class ShallowCopy {
    public static void main(String[] args) {
        Circle c1 = new Circle(new Point(1, 1), 2.0);
        Circle c2 = c1.shallowCopy();
        Circle c3 = c1.deepCopy();

        System.out.println("c1="+c1); // 1 1 2.0
        System.out.println("c2="+c2); // 1 1 2.0
        System.out.println("c3="+c3); // 1 1 2.0
        c1.p.x = 9;
        c1.p.y = 9;
        System.out.println("= c1의 변경 후 =");
        System.out.println("c1="+c1); // 9 9 2.0
        System.out.println("c2="+c2); // 9 9 2.0
        System.out.println("c3="+c3); // 1 1 2.0
    }
}
```



▲ 그림9-1 얕은 복사와 깊은 복사의 비교

c1을 변경했을 뿐인데, c2도 영향을 받는다. 그러나 c3는 전혀 영향을 받지 않는다.



▶ getClass(): 자신이 속한 클래스의 Class객체를 반환하는 메서드. Class객체는 이름이 'Class'인 클래스의 객체로 클래스의 모든 정보를 담고 있으며, 클래스당 1개만 존재한다. 클래스 파일이 '클래스 로더(ClassLoader)'에 의해서 메모리에 올라갈 때, 자동으로 생성된다.

```
public final class Class implements ... { // Class클래스
    ...
}
```

클래스로더는 실행 시에 필요한 클래스를 동적으로 메모리에 로드하는 역할을 한다. 먼저 기존에 생성된 클래스 객체가 메모리에 존재하는지 확인하고, 있으면 객체의 참조를 반환하고 없으면 클래스패스(classpath)에 지정된 경로를 따라서 클래스 파일을 찾는다. 못 찾으면 ClassNotFoundException이 발생하고, 찾으면 해당 클래스 파일을 읽어서 Class객체로 변환한다. 그러니까 파일 형태로 저장되어 있는 클래스를 읽어서 Class클래스에 정의된 형식으로 변환하는 것이다. 즉, 클래스 파일을 읽어서 사용하기 편한 형태로 저장해 놓는 것이 클래스 객체이다. 클래스 파일을 메모리에 로드하고 변환하는 일은 클래스 로더가 한다.

☞ Class객체를 얻는 방법: 클래스의 정보가 필요할 때, 먼저 Class객체에 대한 참조를 얻어와야 한다.

```
Class cObj = new Card().getclass();   // 생성된 객체로부터 얻는 방법
Class cObj = Card.class; // 클래스 리터럴(.class)로부터 얻는 방법
Class cObj = Class.forNmae("Card"); // 클래스 이름으로부터 얻는 방법
```

forName()은 특정 클래스 파일, 예를 들면 데이터베이스 드라이버를 메모리에 올릴 때 주로 사용한다.

Class객체를 이용하면 클래스에 정의된 멤버잉 이름이나 개수 등, 클래스에 대한 모든 정보를 얻을 수 있기 때문에, Class객체를 통해서 객체를 생성하고 메서드를 호출하는 등 보다 동적인 코드를 작성할 수 있다.

```
Card c = new Card(); // new연산자를 이용해서 객체 생성
Card c = Card.class.newInstance(); // Class객체를 이용해서 객체 생성.
// 단, newInstance()는 InstantiationException 예외에 대한 예외처리가 필요하다.
```

```java
final class Card {
    String kind;
    int num;

    Card() {
        this("SPADE", 1);
    }

    Card(String kind, int num) {
        this.kind = kind;
        this.num  = num;
    }

    public String toString() {
        return kind + ":" + num;
    }
}

class ClassEx1 {
    public static void main(String[] args) throws Exception {
        Card c  = new Card("HEART", 3);        // new연산자로 객체 생성
        Card c2 = Card.class.newInstance();    // Class객체를 통해서 객체 생성

        Class cObj = c.getClass();

        System.out.println(c);
        System.out.println(c2);
        System.out.println(cObj.getName()); // Card
        System.out.println(cObj.toGenericString()); // final class Card
        System.out.println(cObj.toString()); // class Card
    }
}
```

Class객체에 대한 참조를 얻으면 Java API문서에 있는 Class클래스의 수많은 메서드를 통해 클래스에 대한 다양한 정보를 얻을 수 있다.

Resources: 자바의 정석