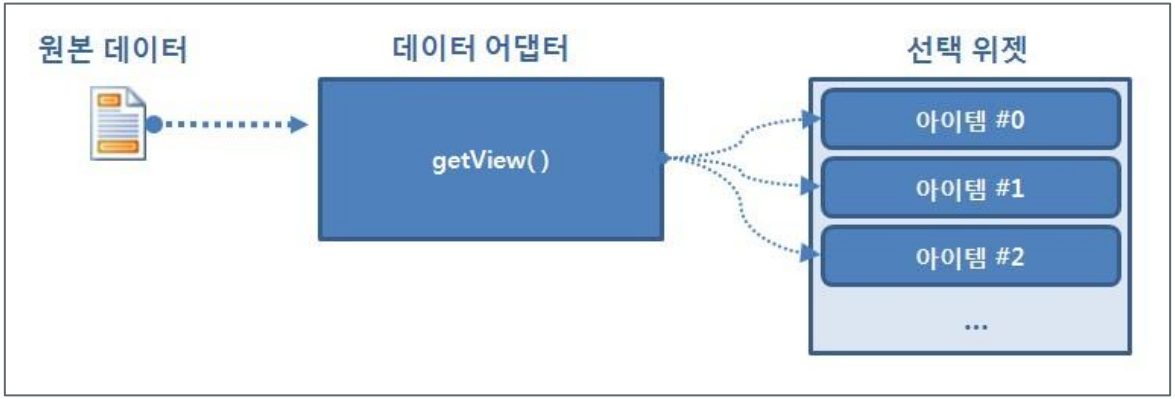
리스트 뷰
(ListView)

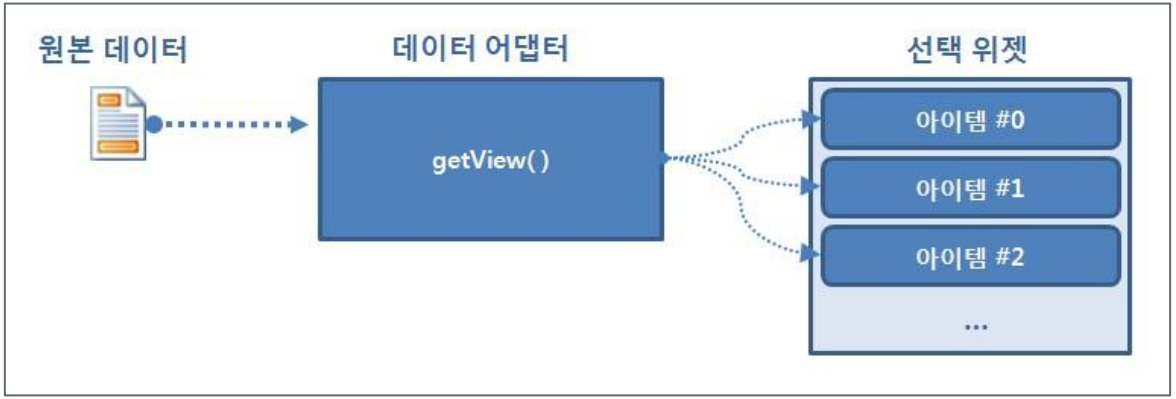
선택 위젯(Selection Widget)이란?



=> 한 위젯에서 여러 데이터를 보여주려고 하는 경우에는 위젯 안에 여러 데이터를 각각의 뷰에 담아 보여주고, 그 중 하나를 **선택**했을때 어떤 아이템이 **선택**되었는지를 확인한 후 이벤터 처리를 위한 코드를 실행해야 한다. 이렇게 **여러 개의 데이터를 보여주고 그 중 하나를 선택**했을때 원하는 기능을 실행할 수 있는 위젯을 **선택 위젯(Selection Widget)**이라 한다.

=> 선택 위젯이 일반 위젯과 다른 점은 **어댑터 패턴**을 사용해야 한다는 점이다.

선택 위젯(Selection Widget)이란?



=> 리스트뷰나 리사이클러뷰와 같은 선택 위젯은 **껍데기**일뿐이고, 어댑터에서 **(1) 데이터를 관리**하고.

(2) 화면에 보여지는 뷰도 어댑터의 getView() 메소드에서 결정한다.

=> 선택 위젯의 가장 큰 특징은 원본 데이터를 위젯에 직접 설정하지 않고 위의 그림처럼 **어댑터**라는 클래스로 사용하도록 되어 있다는 점이다.

리스트 뷰 예시

1. activity_main.xml 에 ListView 추가

```
<ListView
  android:id="@+id/listView"
  android:layout_width="match_parent"
  android:layout_height="match_parent"/>
```

: **listView**는 껍데기일뿐, 데이터 관리와 뷰 생성은 어댑터가 책임진다.

2. MainActivity.java 에 ListView 객체 생성

```
ListView listView = findViewById(R.id.listView);
```

리스트 뷰 예시

3. MainActivity.java 에 SingerAdaper 객체 생성 및 아이템 추가.

```
singerAdapter = new SingerAdapter();
singerAdapter.addItem(new SingerItem("소녀시대", "010-1000-1000", R.drawable.idol));
singerAdapter.addItem(new SingerItem("에이핑크", "010-1234-1234", R.drawable.idol));
singerAdapter.addItem(new SingerItem("트와이스", "010-2297-5762", R.drawable.idol));
singerAdapter.addItem(new SingerItem("레드벨벳", "010-9330-1994", R.drawable.idol));
singerAdapter.addItem(new SingerItem("여자친구", "010-2531-6735", R.drawable.idol));
```

4. MainActivity.java 에서 listView 변수에 setAdapter() 메소드에 singerAdapter를 매개변수로 지정.

```
listView.setAdapter(singerAdapter);
```

리스트 뷰 예시

- SingerAdapter 클래스 살펴보기

```
class SingerAdapter extends BaseAdapter {
    // 데이터 관리 ArrayList
    ArrayList<SingerItem> items = new ArrayList<>();

    // 데이터 추가 메소드
    void addItem(SingerItem item) {
        items.add(item);
    }
```

리스트 뷰 예시

- SingerAdapter 클래스 살펴보기

```
    // 각 아이템의 뷰 생성
    @Override
    public View getView(int i, View view, ViewGroup viewGroup) {
      SingerItemView view2;
      if(view == null){
          view2 = new SingerItemView(getApplicationContext());
      } else {
          view2 = (SingerItemView)view;
      }

      SingerItem item = items.get(i);
      view2.setName(item.getName());
      view2.setMobile(item.getMobile());
      view2.setImageView(item.getResId());

      return view2;
    }
}
```

리스트 뷰 정리

- ListView

  : 리스트 뷰는 여러 개의 아이템을 위 아래로 순서대로 보여주는 선택 위젯으로,

  어댑터를 통해 각 아이템의 데이터와 뷰 객체를 받아 화면에 출력한다.

- Adapter

  : 어댑터는 ArrayList()와 각 메소드로 리스트뷰에 출력될 데이터를 추가 및 관리한다.

  : 어댑터는 getView()라는 메소드를 통해 뷰 객체를 **리스트뷰**에 반환한다.

: 리스트뷰는 화면에 출력하는 역할이고, 어댑터는 데이터 관리 및 뷰 객체를 생성하는 역할이다.

어댑터 패턴

- 어댑터 패턴을 쓰면 좋은 점

=> **한 클래스의 인터페이스를 클라이언트에서 사용하고자 하는 다른 인터페이스로 변환한다. 어댑터를 이용하면 인터페이스 호환성 문제 때문에 같이 쓸 수 없는 클래스들을 연결해서 쓸 수 있다.**

'이미 제공되어 있는 것'과 '필요한 것' 사이의 '차이'를 없애주는 디자인 패턴이 **Adapter 패턴**이다.

Adapter 패턴은 Wrapper 패턴으로 불리기도 한다.

일반 상품을 예쁜 포장지로 싸서 선물용 상품으로 만드는 것처럼, 무엇인가를 한번 포장해서 다른 용도로 사용할 수 있게 교환해 주는 것이 wrapper 이며 adapter 이다.

Adapter 패턴에는 다음과 같이 두 가지 종류가 있다.

- 클래스에 의한 Adapter 패턴 (상속을 사용한 Adapter 패턴)
- **인스턴스에 의한 Adapter 패턴 (위임을 사용한 Adapter 패턴)**

어댑터 패턴

- **어떤 경우에 사용하는 것일까?**

=> 메소드가 필요하면 그것을 프로그래밍하면 되지 ‘ 왜 Adapter 패턴이 필요한 것일까?’

우리는 언제나 처음부터 프로그래밍을 한다고 할 수는 없다.
이미 존재하고 있는 클래스를 이용하는 경우도 자주 있다.
특히 그 클래스가 충분한 테스트를 받아서 버그가 적으면 실제로
지금까지 사용된 실적이 있다면 어떻게든 그 클래스를 부품으로 재이용하고 싶을 것이다.

Adapter 패턴은 기존의 클래스를 개조해서 필요한 클래스를 만든다.

- 이 패턴으로 필요한 메소드를 발빠르게 만들 수 있다.
- 만약 버그가 발생해도 **기존의 클래스에는 버그가 없으므로 Adapter 역할의 클래스를 중점적으로 조사하면 되고, 프로그램 검사도 상당히 쉬워진다.**

어댑터 패턴

- **비록 소스가 없더라도 ...**

이미 만들어진 클래스를 새로운 인터페이스에 맞게 개조시킬 때는 당연히 Adapter 패턴을 사용해야 한다. 그러나 실제 우리가 새로운 인터페이스에 맞게 개조시킬 때는 기존 클래스의 소스를 바꾸어서 '수정'하려고 생각한다.
'이것을 조금 바꾸면 분명 작업은 끝이다' 라고 생각하기 쉽다.
그러나 그렇게 하면 동작 테스트가 이미 끝난 기존의 클래스를 수정한 후에 다시 한번 테스트해야 한다.

Adapter 패턴은 **기존의 클래스를 전혀 수정하지 않고** 목적한 인터페이스에 맞추려는 것이다.
또한 Adapter 패턴에서는 기존 클래스의 소스 프로그램이 반드시 필요한 것은 아니다.
기존 클래스의 사양(Interface)만 알면 새로운 클래스를 만들 수 있다.

어댑터 패턴 예시

- 기존의 인터페이스와 클래스

```
public interface APlayer {

   void play(String fileName);
   void stop();
}

public class APlayerImpl implements APlayer {
   @Override
   public void play(String fileName) {
      System.out.println("(A) " +fileName);
   }

   @Override
   public void stop() {

   }
}

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

      // 기존에 사용하던 방식
      APlayer player1 = new APlayerImpl();
      player1.play("aaa.mp3");
   }
}
```

: 인터페이스 변수 player1 에 APlayerImpl 객체를 생성하는 기존의 방식.

어댑터 패턴 예시

- 새로 도입한 인터페이스와 클래스

```
public interface BPlayer {

   void playFile(String fileName);
   void stopFile();
}

public class BPlayerImpl implements BPlayer {
   @Override
   public void playFile(String fileName) {
      System.out.println("(B) " + fileName);
   }

   @Override
   public void stopFile() {
   }
}

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

      // BPlayer : 새로 도입된 방식(잘 동작할 것이다.)
      BPlayer player2 = new BPlayerImpl();
      player2.playFile("bbb.mp3");
   }
}
```

: 인터페이스 변수 player2 에 BPlayerImpl 객체를 생성하는 새로운 방식.

어댑터 패턴 예시

- 어댑터 이용하여 기존의 인터페이스 변수로 새로운 클래스 객체 사용.

```
public class BToAAdapter implements APlayer {


  private BPlayer media;
  public BToAAdapter(BPlayer media) {
     this.media = media;
  }
  @Override
  public void play(String fileName) {
     // A의 메서드로 B의 메서드를 호출
     System.out.println("Using Adapter : ");
     media.playFile(fileName);
  }
  @Override
  public void stop() {

  }
}

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

     // 어댑터 사용.
     player1 = new BToAAdapter(new BPlayerImpl());
     player1.play("ccc.mp3");

  }
}
```

어댑터 패턴 정리

- 기존의 잘 동작하던 코드와 새로 도입된 코드를
  변경없이 사용하고 싶을 때 **어댑터 패턴**을 이용한다.
  어댑터 적용 후 에러가 난다면 **어댑터 부분만 보면 될 것이다.**

- 위의 예시의 코드는 APlayer 인터페이스에 맞춰서 코딩되어있다.
  그러므로 여기서 APlayer에 대입되는 객체만 수정해주면
  APlayer 인터페이스가 사용되는 부분에서는 수정할 필요가 없다.