ジェネリクスというのが、

型をパラメーターとして抽象的に扱う事なんですよね。

分かりづらいですが今は聞き流してください。

では、実際にコレクションでジェネリクスを使ってみましょう。

Listを作成する例で行きましょう。

Listを生成するときは<>で型情報を付与できますよね。

これによって追加する要素の方がStringであればString型しか入れられなくなりますよね。

しかしこれがジェネリクスというやつですね。

import java.util.List;

import java.util.ArrayList;

List<String> items = new ArrayList<String>();

ここで＜＞に囲まれた型を「型パラメータ」と呼びます。

型パラメーターにはオブジェクト型ではなくて、プリミティブ型を用い

らないといけないんでしたよね。

intであれば、Integerとなるし

でさっきも言った通り、この状態で、String以外を入れようとするとコンパイルエラーになります。

items.add("bag"); //これはOKですね

items.add(0); //これはエラー

items.add(true); //これもエラー

この要素を取り出して他の変数に入れる時も、String型であることが

保証されているため、他の方の変数に代入もできません。

String item = items.get(0); //これはOKですね

int item = items.get(0); //error

boolean item = items.get(0);

上記のような挙動は一見当たり前のように思えるかもしれませんが、

ジェネリクスの機能が無かったらどうなるのか考えてみましょう。

型を宣言せずにListオブジェクトを作くるとどうなるでしょうか？

型を宣言しなかった場合、全てのクラスの親クラスであるObject型

の要素を持ったListが生成されます。

つまり、Stringも入れられるし、Integerでも行ける

List items = new ArrayList();

上記をジェネリクスを使ってあえて書いてみると

List<Object> items = new ArrayList<Object>();

となるのです。

Excelで言うと、バーリアン型ですよね。

どんな型でも入れられるのですが、期待している型以外が呼び出されたり

代入されてしまうこともあり、コンパイルエラーせずに挙動してしまうので、

期待する結果と異なる場合、どこが問題なのかを見つけにくいという問題が

がありますよね。

でこのいろいろな方が代入できるList<Object>から値を取り出すときは

キャストを行って型を指定して取り出します。

String item1 = (String)items.get(0);

ちなみに、親クラスから、子クラスを指定して肩を取り出すことをダウンキャストというのに対して

子クラスを親クラスに戻して呼び出すことをアップキャストと言います。

でこの問題点としては、呼び出すときに毎回キャストするのが面倒という事と、

実際にそれがどんな値なのかという事がわからなくなってしまいます。

List<Object>として変数を宣言すると、初めてそれを見た人は

どんな用途で、使われるのかがわかりません。intも入るし、Stringも入るのでわかりづらいです。

それに、別のところで値が呼び出されるときにダウンキャストするときも

ダウンキャストするまで、適切な型が格納されているのかがわかりません。

実際に次の例を見てみましょう。

String item2 = (String)items.get(1);

Stringの変数にitemsからStringの要素を取り出して格納するので、

コンパイル時にはエラーが起きませんが、実行時にエラーが発生します。

実はこのitems.get(1)にはInteger型の要素が格納されていた場合にそうなります。

コンパるえらーには引っかからずに、実行しないとエラーが起きるかわからないことになりますし、

かなり不安なプログラムになりますよね。

そういった型の相違を事前にコンパイルエラーで検出できるのがジェネリクスという事です。

ですので、Listなどを宣言するときは、ジェネリクスを使うことが安全だとこれで

分かりましたよね。

まとめ

＞ジェネリクスとは？

型をパラメーターとして抽象的に扱うことで、汎用的に扱うことができる機能

で、プログラムを汎用的に用いるための仕組みという事でしたが、

Listだけではメリットがまだ発揮できていないのです。

では、実際に汎用的とはどういうことかを見ていきましょう。

Basketというクラスを作り、特定の型を1つだけ入れられるようにします。

型は変更できないようにします。

class Basket<E> {

private E elem;

Basket(E e) {

elem = e;

}

void replace(E e){

elem = e;

}

E get() {

return elem;

}

}

このEに入るのが別のクラスのパラーメータです

オリジナルのAppleというクラスを作っておいて、

Basket<Apple>という風にパラメーターを入れられます。

EにAppleという型がパラメーターとして設定された場合に、

Basketクラス内にあるEという変数は全てAppleクラスとして置き換えられます。

これは、別のクラスを代入しても可能です、

つまり、他のクラスを間違って入れられないようにするために、

クラスの定義と同時にパラメーターとして方も入れるように設定をすれば

ジェネリクスの仕組みにできます。

このように、クラス内で、別のクラスの型を扱う挙動がある場合は、

ジェネリクスにしておく方が安心です。

MapやListも同じような仕組みですね。

次に境界型パラメータというのがあります。

型パラメーターとして受け取れるクラスを、