Java types

Primitive – int, long, short, byte, char, Boolean, float, double

Reference – everything else

Java generics is java’s parametric polymorphism

- Idea to allow type (integer, string, etc and user defined types) to be a parameter to methods, classes and interfaces.

e.g. Public static <A> A choose (Boolean b, AX, AY)

Advantages

- type safety – can hold only a single type of objects in generics

- Don’t have to type cast

- errors appear at compile time, rather than at runtime

e.g. create an ArrayList to store names of students but add an integer, allowed by compiler, but causes runtime error.

ArrayList al = new ArrayList();

Al.add(“hello”);

Al.add(“goodbye”);

Al.add(10); // compiler allows this

String s3 = (string) al.get(2) //causes java.lang.castclass exception

ArrayList <String> al = new ArrayList<String>(); //creates an ArrayList with string specified

Al.add(“hello”);

Al.add(“goodbye”);

Al.add(10); // compiler does not allow this

Static vs dynamic typing

Static – type determined at compile time (Haskell, Java)

Dynamic – type determined at runtime

Type variable can stand for a reference type only

* Primitive types have a wrapper type e.g. Int and Integer
* Can pass wrapper type into a generic
* Boxing/unboxing converts from/to a primitive type

Subtyping

T1 <: t2 “t1 is a subtype of t2”

* Everywhere a value of type t2 is excepted a value of t1 is acceptable
* Subtyping relation is reflexive and transitive
* For all t, if t is a reference type, then t <: object
* Can’t create arrays of generic type, can use ArrayList

C is covariant when t1 <: t2 implies c<t1> <: c<t2>

C is invariant when t1 <: t2 implies nothing about c<t1> and c<t2>

C is contravariant when t1 <: t2 implies c<t2> <: c<t1> - NOT A RULE IN JAVA

In java, arrays are covariant, everything else is invariant

Covariant types are read-only and except subtypes

Contravariant types are write-only and accept supertypes

e.g Feline a subtype of animal

List<Feline> felines = new ArrayList<Feline>();

Public Static void flist (List<Animal> la) {

}

flist(felines) // not allowed, ArrayList<feline> not a subtype of List<Animal>

* Generic types not covariant

Public Static Void fArray (Animal [] aa) {

}

Feline[] felineArray = new Felineb[8];

fArray(felineArray); // allowed, Feline[] a subtype of animal[]

* Generics used for tighter type checks at compile time
* Java compiler applies type erasure
  + Compiler replaces generic parameter with actual class or bridge method
  + Compiled generic code using Java.lang.object wherever T is

Public static void fArray(Animal [] aa) {

Aa[0] = newLittleDoggy)(); //causes ArrayStoreException

Public static void fList (List<? extends Animal> la) {

For (Animal a: la) {

a.breathe();

}

flist(felines); // allowed, can pass a list of felines b/c method expects animal or anything that is a subclass

la.add(new littledoggy) // not allowed, can’t add to the list b/c don’t know what type of animal is in the list

Public static void contraList (list<Tiger> input)

Contralist(felines); // cannot do this list <feline> not a super type of list<tiger>

Public static void contraList(list<? super Tiger> input) {

Contralist(felines) // function can receive list of tiger and list of anything that is a super type of tiger

Object obj = input.get(0); // have to use object, can’t be sure what type element is allowed

Input.add(new Tiger()); // allowed

Input.add(new Feline()); // not allowed, could be given a list of tiger

“Producer extends, consumer super”.

A producer-like object that produces objects of type T can be of type parameter <? extends T>, while a consumer-like object that consumes objects of type T can be of type parameter <? super T>.

ArrayList<? extends Number> nums = ints // can contain any object that is either of a number type or its subtype

Number n = nums.get(0); // we know that whatever we get from ArrayList can be upcasted to a Number type

nums.add(new Integer(2))); // not ok, we cannot be sure of the “actual type” of the object. All we know is that it must be a number or its subtypes

ArrayList<? super Integer> nums = ints;

Nums.add(new Integer(2));//certain whatever “actual type” of the object is, it must be Integer or is supertype, and thus accept an Integer object.

Integer n = nums.get(0); // can’t be sure that we will get an integer, nums could be referencing an ArrayList of objects.

Object I = Integer.valueof(42);

String s = (String)I; //ClassCastException thrown, string not a subclass of integer

public static void main(String[] args)

{

Object[] strings = new String[5];

strings[0] = 5; // will throw ArrayStoreException

}

Wildcards

* Question mark (?) known as wildcard in generic programming, represents an unknown type
* Upper bounded wildcards, used when you want to relax the restrictions on a variable
  + e.g. a method that works on list<integer>, list<double> and list<number>, use upper bounded wildcard.

Public Static void add(list<? Extends Number> list) // says that ? has to be a number or a subtype of number

* Lower bounded wildcards – expressed by using wildcard (‘?’) followed by the super keyword, followed by its lower bound <? Super A>
  + e.g Collectiontype <? Super A>
  + e.g. Public Static Void printonlyintegerclassorsuperclass (list? Super Integer>) // says that ? has to be integer or a superclass of integer
* Unbounded Wildcard – specified using wildcard character (?)
  + Useful when:
    - Writing a method which can be employed using functionality provided in object class
    - When the code is using methods in the generic class that don’t depend on the type parameter
  + e.g. private static void printlist(List<?> list)

Type equivalence

* Nominal typing (types known by their name) – Haskell, Java, C, C++
  + e.g. TYPE T1 = ARRAY [0..10] OF CHAR;
  + TYPE T2 = ARRAY [0..10] OF CHAR;
  + Types T1 and T2 are not equivalent.
* Structural typing (types known by their structure) – ML’s records
  + TYPE T1 = RECORD
  + a:INTEGER;
  + b:ARRAY [0..10] OF CHAR;

END;

TYPE T2 = RECORD

a:INTEGER;

b:ARRAY [0..10] OF CHAR;

END;

* + Types T1 and T2 are equivalent.

Type conversions

* Widening primitive conversion
  + e.g. double d = 5;

Optional - A container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value.

* A wrapper class that contains an optional value, meaning it can either contain an object or can be empty.

Optional<user> emptyOpt = Optional.empty(); // returns an empty instance of Optional

Optional<user> opt = Optional.of(user); // creates an optional object that can contain a value, should only use it when sure the object is not null

Optional<user> opt = Optional.ofNullable(user);// used if object can be both null or not-null

ofElse() method returns the value if its present, or the argument it receives if not

e.g. User result = Optional.ofNullable(user).orElse(user2);

The intended use of *Optional* is mainly as a return type. After obtaining an instance of this type, you can extract the value if it’s present or provide an alternate behavior if it’s not.