Generic type cannot be used with static context

Private static T t; 🡪 not allowed

Type Erasure: generics is used only at compile time. Compiler will erase the generic type during compilation. Store<T> will be replaced with Store<Object>

Explicit Casting: When Store<String> is created, get methods will be automatically casted from object to String.

Type argument cannot be primitive. Store<int> is not allowed.

A screenshot of a cell phone

Description automatically generated

Type parameter with one or more bounds

<TypeParameter extends bound1 & bound2 & ….>

GenericsDemo<T extends List>

GenericsDemo<List> 🡪fine

GenericsDemo<ArrayList> 🡪 fine

GenericsDemo<LinkedList> 🡪 fine

GenericsDemo<Collection> 🡪 compilation error

Bounded Type parameter can access methods defined by bounds

A screenshot of a cell phone

Description automatically generated

Valid Bounds:

Class

Interface

Enum

Parameterized Type <T extends Comparable<T>>

Invalid Bounds:

Primitives

Arrays

1. **Type argument must be subtype of all bounds**

Class GenericsDemo <T extends List & Serializable> {}

GenericsDemo<List> 🡪 not valid because List doesn’t implement Serializable

GenericsDemo<ArrayList> 🡪 Fine ArrayList implements both List and Serializable

1. If class is one of the bounds, it must be first
2. First bound is class 🡪 remaining must be interfaces (only one class allowed)
3. Final classes & enums 🡪 type argument is bound itself

A screenshot of a cell phone

Description automatically generated

**Unbounded Wildcard**

Class Store<T> T is type parameter and it is unknow type.

1. Void go(Store<?> someStore) {} ? is unbounded wildcard
2. Wildcard can be used only as Type Argument
   1. Class Test<?, ?> {

Private ? a;

}

1. Common ~ void go(Store<?> someStore)
2. Not common!!
   1. Store<String> strStore = new Store<>();
   2. Store<?> someStore = strStore;
3. Why not Store<Object> instead of Store<?> ?
   1. Store<Object> someStore = new Store<Object>();
   2. Store<?> someStore = new Store<String>();
   3. Store<?> someStore = new Store<Integer();
4. Cannot invoke methods that use class-level type parameters with any arguments except null.

**Generics and Invariance**

Generics Promise 🡪 compile-time type safety

Book is a subtype of Bookmark, but Store<Book> is not subtype of Store<Bookmark>

List<Bookmark> 🡪 List<Book> 🡪 Not Allowed.

List<Bookmark> 🡪 ArrayList<Book> 🡪 Allowed

Why we need Invariance…

void go(ArrayList<Bookmark> items) {

//if invariance is not present, lets say use passes Book which is subclass of Bookmark

//and method allows it, then the program can try to add Movie which is a subclass of

//Bookmark.

Items.add(new Movie()); //now items have both books and movies which is bad

}

go(new ArrayList<Book>());

So invariance is required for compile time type safety.

**Arrays are Covariant**

Bookmark 🡪 Book

Bookmark[] 🡪 Book[] 🡪 Allowed.

Bookmark[] items = new Book[1];

Items[0] = new Movie(); // ArrayStoreException (Runtime) //no compile type safety hehehe

Generic Methods:

Compile-time type safety + implicit casts

Both static and instance method can be generics.

Static utility methods are perfect candidates for generice methods.

Methods and constructors can have own **type parameters**

{modifier} <T1, T2, … > returnType methodName(T1 param1, T2 param2, ….) {…..}

<T> T[] toArray(T[] a); //java.util.Collection

static <T> Boolean replaceAll(List<T> list, T oldVal, T newVal) {….}

class GenericsDemo <T> {

<T> void go(T object) { …. }

}

class GenericDemo <E> {

<T> void go(T obj1, E obj2) { …. }

//class level type parameters (E in this case) cannot be used in static method.

//this method cannot be static.

}

<T extends List & Serializable> void go(T object) { …. }

<T extends E> void go(T object) { … }

<T1, T2 extends T1> void go(T1 obj1, T2 obj2) { … }

T 🡪 method level parameter

E 🡪 Class level parameter

**Enclosing type need not be generic:**

**Method Invocation: Type Argument Inference**

**<T> T go(T object) { return object; }**

**Doubl eval = go(1.0); type arg ~ Double 🡪 compiler inference the type**

**String val = go(“java”);**

<T> List<T> emptyList(); 🡪 Return type infered/

Double val = go(“java”); //compile time error.

**Inferring Most Specific Type**

<T> T go(T a1, T a2) { …. }

Serializable s = go(“d”, new ArrayList<String>());

Method Invocation: Explicit Type Argument Specification

Class GenericsDemo {

<T> void go(T object) { …. }

}

GenericsDemo gd = new GenericsDemo();

gd.<Double>go(1.0);

this.<Double>go(1.0);

GenericsDemo.<String>go(“java”);