Lesson-10- Class Notes

Java Generics introduced from JDK 5,

Pre JDK --> Java supports only Raw Type - Object

The type Erasure concept is introduced to support backward compatibility.

```
public static void print(List<String> stringList) {
                                System.out.println("List of Strings");
                        }
public static void print(List<Integer> integerList) {
                                System.out.println("List of Integers");
                        }
```

Generic Subtyping Is Not Covariant.

Simple Analogy: Consider a box of apples (Box<Apple>) and a box of fruits (Box<Fruit>). Even though apples are fruits, a box of apples is not the same as a box of fruits.

```
List<Integer> integerList = new ArrayList<>();
List<Number> numberList = integerList; // This is illegal in Java.
//numberList.add(3.14); // This would be problematic if the above line were allowed.
```

If the assignment were allowed, numberList would be pointing to the same list as integerList.

Adding a Double to numberList would then actually add a Double to integerList, which should only hold Integer values, thus breaking the type safety that generics should provide.

By disallowing such assignments, Java ensures that collections' integrity and type safety are maintained.

Wild card Boundaries are,

1. ? extends Parent Type (Upper Boundary)
2. > super Child Type (Lower Boundary)
3. ? (No boundaries) - Unknown type. You don't care about the specific type of parameter
   --> This is useful when you are not dependent on any input arguments
   and return type.
   Example: size(), clear();
   Look at the Collections library to see examples from <?>

Limitations of ? extends wildcard,

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
```

```
List<? extends Number> nums = ints; // Will be fine
nums.add(3.14);      //compiler error
```

The compiler error occurs because 3.14 is a Double, and although Double is a subtype of Number, the numbers list is not necessarily a List<Double>; it could be a List<Integer> or a List<BigDecimal> or any other subtype of Number if the compiler allowed you to add a Double to numbers which breaking the type safety guarantee.

Simple Analogy: Let's think of a List in Java as a box where you can put things in and take things out. Now, when you have a List that uses ? extends Something, it's like having a special box that only allows certain kinds of "Somethings" to be taken out, but you're not allowed to put anything into it.

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);

List<? extends Number> nums = ints;
```

The compiler verifies that:

- The right side of the assignment (ints) is a List of something that is a Number or extends Number.
- The left side (nums) is declared to hold a list of any type that fits the criteria of extending Number.

Both the left and right sides are satisfied for the type safety at compile time.

What's important to note is that after this assignment, you cannot add elements to nums. Even though you know nums references a List<Integer>, the compiler only knows it as a List of "some subclass of Number" and therefore won't allow additions to prevent possible type safety violations.

? super wildcard

```
// Something has a lower boundary of sub type Integer
                List<? super Integer> list = new ArrayList<>();
                list.add(10);

                int x = list.get(0); // Compilation error
                int x =(int) list.get(0); // Will be fine
```

So, ? super is used when you want to work with a collection that can hold elements of a certain type or any of its super classes. You can always add an Integer to List<? super Integer> because Integer is guaranteed the list is prepared to hold. But when you take something out, Java treats it as an Object, which is the most general type, to ensure you don't make a mistake about what you're getting.