Subtyping & Java generics

Generics

- Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming.
- A *generic type* is a generic class or interface that is parameterized over types.
- A *generic class* is defined with the following format:
  - class name <T1,T2,….Tn> { /*…*/ }
  - The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn

Generics

- Without generics type checking will be performed at run time instead of compile time.
- With generics the type checking happens at compile time.
- Improves readability and robustness
- Familiar classes and interfaces that use generics in java: Most of the classes and interfaces in Collections like Collection, List, HashMap,Set, HashSet, ArrayList etc.
- In fact, java.lang.Class is generic too!
- Type parameters are analogous to the ordinary parameters used in methods or constructors.
- <T> is like a formal parameter
- ClassName <Object> is not the super type of class ClassName
- Supertype of all kinds of collections: Collection <?>

Method declarations can be Generic too!

- Like the type declarations, method declarations can be generic too and can be parameterized by one or more type parameters.
- *Generic methods* are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.
- The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type

Generic methods

- When calling a generic method you don’t have to pass an actual type argument to a generic method.
- Based on the types of the actual arguments the compiler infers the type of the generic method

Raw Types

- A *raw type* is the name of a generic class or interface without any type arguments
- Example:

  public class Shape<T>{ /*..*/}

  To create parametrized type of Shape <T>, supply an actual type argument for the formal type parameter T.

  Shape<Integer> s=new Shape<Integer>();

  If the actual type is omitted, you create a raw type of Shape<T>,:

  Shape s=new Shape();
- Note: A non-generic class or interface type is not a raw type.

Bounded Type Parameters

- Times when you want to restrict the types that can be used as type arguments in a parameterized type.
- Syntax of bounded type parameters is:
- List the type of parameters name, followed by extends keyword, followed by its upper bound.
  - <T extends Integer>
- It is possible to have multiple bounds:
- Syntax: <T extends S1 & S2 & S3>
- A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first else you will get a compile time error.

Bounded wildcards

- ? Represents an unknown type and is called the wildcard.
- Situations where ? is used:
  - type of a parameter
  - Field
  - local variable;
  - sometimes as a return type (though it is better programming practice to be more specific).

Bounded Wildcards (Upper bound)

- Bounded wildcards:
  - <? extends Supertype>
    - ? Means unknown type
    - But <? extends Supertype> implies supertype is the upper bound for the wildcard.
    - When <? extends Supertype> is used as the formal parameter yo
    - <? extends T> : wildcard used for read only data structure
    - upper bounded wildcard to relax the restrictions on a variable.

Bounded Wildcards (Lower Bound)

- <? super T> :
  - ? means of unknown type and super T implies a lower bound for the wildcard.
  - *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type.
  - wildcard used for write only data structure
- You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

Unbounded Wildcards

- <?> :
  - ? means of unknown type.
- Scenarios where unbounded wildcard is useful:
  - If you are writing a method that can be implemented using functionality provided in the Object class.
  - When the code is using methods in the generic class that don't depend on the type parameter. For example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

Erasure

- Generics are implemented by the Java compiler as a front-end conversion called erasure.
- Erasure gets rid of (or erases) all generic type information at run time.
- All the type information between angle brackets is thrown out. For example:
    - a parameterized type like List<String> is converted into List

Restrictions on Generics

- Cannot Instantiate Generic Types with Primitive Types
- You cannot create an instance of a type parameter.
- Cannot Declare Static **Fields** Whose Types are Type Parameters
  - Why? Because class's static field is a class-level variable shared by all non-static objects of the class.
- You cannot create arrays of parameterized types.
- A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
public void print(Set<String> strSet) { }
public void print(Set<Integer> intSet) { } }
```

Why? The overloads would all share the same classfile representation and will generate a compile-time error.