Generics

- Helps prevent `ClassCastException` at runtime when working with collections.
- Built for type checking at compile time.
- Generic types used *without* any type arguments are called *raw types*.
- Type arguments are associated with *expressions/variables* not *instances*.

Type Erasure

- Compiler replaces the type parameters in bytecode:
  - **Unbounded type** parameters are replaced with `Object`, e.g. `List<String>` becomes `List<Object>`
  - **Bounded type** parameters are replaced with its first bound. e.g. `T extends Comparable<T>` becomes `Comparable`
- Inserts necessary type casts for type safety ***if*** the value is used, e.g. assigned, printed, passed to another method, etc. For example, if the result from `<list>.get()` is ignored, no cast is generated.
- No runtime specific type arguments of a generic type at runtime, e.g., `<object_reference> instanceof List<String>` is not allowed.
- Cannot create new instances of a type parameter directly, e.g., `new T()`.

Generics vs Array

- Generics is *compile time* construct, while array is *runtime* construct.
- Which means that a declared generic type *cannot* be cast into a different type argument like this:

  ```
  List<Integer> integerList = new ArrayList<>();
  List<Number> numberList = (List<Number>) integerList; // Not compilable
  ```

- But the runtime information on the actual parameterized type is actually absence as `List`.
- While an array can be cast into wider array type like this:

  ```
  Integer[] integerArr = {1, 2, 3};
  Number[] numberArr = (Number[]) integerArr; // Compilable, redundant cast
  ```

- But the runtime of the actual array instance is always array of `Integer`

  In other words, Arrays are not statically sound but are dynamically checked. While generics are statically sound and not dynamically checked.

See this Stack Overflow thread for Generics array topic.