Generics

- generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods and it was introduced in Java 5.
- type parameters provide a way for you to re-use the same code with different inputs.
- also provide compile-time type safety that allows programmers to catch invalid types at compile time.
- removes risk of ClassCastException at runtime

Generic Methods:
Static and non-static generic methods are allowed, as well as generic class constructors.
Following are the rules to define Generic Methods:
- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments

eg: public static < E > void method_name(E arg)
Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Type witness:
to invoke the generic method addBox, you can specify the type parameter with a type witness as follows:

```
BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
```

Alternatively, if you omit the type witness,a Java compiler automatically infers (from the method's arguments) that the type parameter is Integer:

```
BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
```

Generic Classes:
- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.
- the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

eg: public class class_name<T>

benefits of generics:
- Stronger type checks at compile time.
- Elimination of casts. avoids classCastException
- Enabling programmers to implement generic algorithms.

difference between formal parameters and generic type parameters:
Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Generics Type Naming Convention:
E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
K – Key (Used in Map)
N – Number
T – Type
V – Value (Used in Map)
S,U,V etc. – 2nd, 3rd, 4th types

Raw Types:
A raw type is the name of a generic class or interface without any type arguments.
If the actual type argument is omitted, you create a raw type
eg: Box<Integer> intBox = new Box<>() => Box rawBox = new Box() → raw type.
However, a non-generic class or interface type is not a raw type.

For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

Bounded Type Parameters:

- There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter.
- To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

eg: public static <T extends Number> void maximum(T x)

Multiple Bounds
The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have multiple bounds:

```
<T extends B1 & B2 & B3>
```

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.

Generics, Inheritance, and Subtypes:
Given two concrete types A and B (for example, Number and Integer), MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related. The common parent of MyClass<A> and MyClass<B> is Object.

Type Inference:
Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned. Finally, the inference algorithm tries to find the most specific type that works with all of the arguments.

Type Inference and Generic Methods (The Diamond):
In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called the diamond.

Target Types:
The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation. The target type of an expression is the data type that the Java compiler expects depending on where the expression appears.
eg: static <T> List<T> emptyList();

Consider the following assignment statement:

```
List<String> listOne = Collections.emptyList();
```

This works in both Java SE 7 and 8.

Consider the following method:

```
void processStringList(List<String> stringList) {
 // process stringList
}
```

In Java SE 7, the following statement does not compile:

```
processStringList(Collections.emptyList());
```

Alternatively, you could use a type witness and specify the value as String

```
processStringList(Collections.<String>emptyList());
```

This is no longer necessary in Java SE 8.

Wildcards

- In generic code, the question mark (?), called the wildcard, represents an unknown type.
- The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific).

Upper Bounded Wildcards:

- You can use an upper bounded wildcard to relax the restrictions on the unknown type to be a specific type or a subtype of that type.
- To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its upper bound. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

eg: `public static void process(List<? extends Number> list) { /* ... */ }`

Unbounded Wildcards

The unbounded wildcard type is specified using the wildcard character (?), for example, List<?>. This is called a list of unknown type.

There are two scenarios where an unbounded wildcard is a useful approach:

- If you are writing a method that can be implemented using functionality provided in the Object class.

eg: `public static void printList(List<Object> list) {}` instead we can use `public static void printList(List<?> list) {}`

- 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.

Lower Bounded Wildcards:

- a lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type.
- A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its lower bound: <? super A>.

Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

Guidelines for Wildcard Use:
think of variables as providing one of two functions:

- **An "In" Variable**

An "in" variable serves up data to the code. Imagine a copy method with two arguments: copy(src, dest). The src argument provides the data to be copied, so it is the "in" parameter.

- **An "Out" Variable**

An "out" variable holds data for use elsewhere. In the copy example, copy(src, dest), the dest argument accepts data, so it is the "out" parameter.

- An "in" variable is defined with an upper bounded wildcard, using the extends keyword.
- An "out" variable is defined with a lower bounded wildcard, using the super keyword.
- In the case where the "in" variable can be accessed using methods defined in the Object class, use an unbounded wildcard.
- In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

Type Erasure:

- To implement generics, the Java compiler applies type erasure to Generic Types
- During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded, or Object if the type parameter is unbounded.

eg: public class Node<T extends Comparable<T>> is replaced with Comparable while public class Node<T> is replaced with Object

Erasure of Generic Methods
The Java compiler also erases type parameters in generic method arguments.

Bridge Methods
When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, called a bridge method, as part of the type erasure process.

eg:

```
public class Node<T> {
        public void setData(T data) {}
}
public class MyNode extends Node<Integer> {
        public void setData(Integer data) {}
}
```

After type erasure, the method signatures do not not match.
public void setData(Object data) {} => public void setData(Integer data)

To solve this problem and preserve the polymorphism of generic types after type erasure, a Java compiler generates a bridge method to ensure that subtyping works as expected. For the MyNode class, the compiler generates the following bridge method for setData:

```
// Bridge method generated by the compiler
public void setData(Object data) {}

public void setData(Integer data) {}
```

useful if: MyNode node = new Node<>();
node.setData(Double); → calls method of Node<T>

Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types - Pair<int, char>
- Cannot Create Instances of Type Parameters

```
public static <E> void append(List<E> list) {
        E elem = new E();  // compile-time error
        list.add(elem);
}
```

but can use

```
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
        E elem = cls.newInstance();   // OK
        list.add(elem);
}
and call
append(ls, String.class);
```

- Cannot Declare Static Fields Whose Types are Type Parameters - static T var
- Cannot Use Casts or instanceof with Parameterized Types

```
public static <E> void rtti(List<E> list) {
if (list instanceof ArrayList<Integer>) {  // compile-time error
        // ...
        }
}
```

The most you can do is to use an unbounded wildcard to verify that the list is an ArrayList:

```
public static void rtti(List<?> list) {
        if (list instanceof ArrayList<?>) {  // OK; instanceof requires a reifiable type
                // ...
        }
}
```

- Cannot Create Arrays of Parameterized Types

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // compile-time error
```

- Cannot Create, Catch, or Throw Objects of Parameterized Types - catch (T e), class Parser<T extends Exception>, class MathException<T> extends Exception
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

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) { }
}
```

Can Anonymous inner class be made generic?

No.