- **Generic Class**

We want to write Java classes that have general applicability. For example, a generic Stack, or a generic Bag that can hold different object types. Generics also enable you to detect errors at compile time rather than at runtime. You have used a generic interface Comparable in the programming assignment. Generics let you **parameterize types**. With this capability, you can define **a class** or **a method** with generic types that the compiler can replace with concrete types.

The **key benefit** of generics is to enable errors to be detected at compile time rather than at runtime. A generic class or method permits you to specify allowable types of objects that the class or method can work with. If you attempt to use an incompatible object, the compiler will detect that error.

```
package java.lang;

public interface Comparable<T> {
    public int compareTo(T o)
}
```

Here, **<T>** represents a formal generic type, which can be replaced later with an actual concrete type. Replacing a generic type is called a **generic instantiation**. By convention, **a single capital letter** such as **E** or **T** is used to denote a formal generic type. To see the benefits of using generics, let us examine the code below. The statement in (a) declares that c is a reference variable whose type is Comparable and invokes the compareTo method to compare a Date object with a string. The code compiles fine, but it has a runtime error because a string cannot be compared with a date.

The statement in (b) declares that c is a reference variable whose type is Comparable and invokes the compareTo method to compare a Date object with a string. This code has a compile error because the argument passed to the compareTo method must be of the Date type. Since the errors can be detected at compile time rather than at runtime, the generic type makes the program more reliable.

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a)

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b)

Generic types must be reference types. You cannot replace a generic type with a primitive type such as int, double, or char. You need to use the wrapper classes in this case; for example, Integer class. A generic type can be defined for a class or interface. A concrete type must be specified when using the class to create an object or using the class or interface to declare a reference variable.

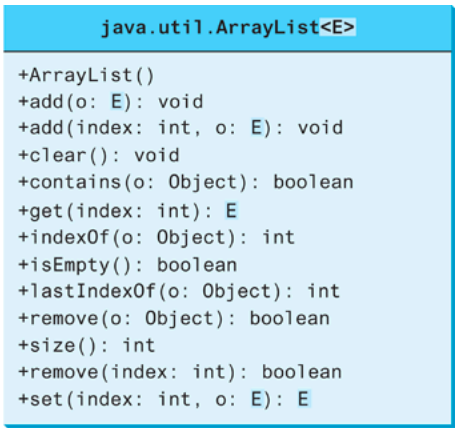
```
public class GenericStack<E> {
private java.util.ArrayList<E> list = new java.util.ArrayList<>();
...
}
```

Creating a stack to hold strings:

```
GenericStack<String> stack1 = new GenericStack<>();
GenericStack<Integer> stack2 = new GenericStack<>();
```

To define a generic type for a class, place the <E> or <T> after the class name, such as the GenericStack class above.

As another example, ArrayList class is a container class in the Java Collections; it is a **generic class**.



The statement: **ArrayList<String> list = new ArrayList<>();** create an instance of ArrayList with the String type. You can only add String objects to the list. With the statement

ArrayList<Employee> elist = new ArrayList<>();,

you can only add Employee objects to the list. As a result, **casting is not needed** to retrieve a value from a list with a specified element type because the compiler already knows the element type. If the elements are of wrapper types, such as Integer, Double, and Character, you can directly assign an element to a primitive-type variable. This is called auto unboxing. For example,

ArrayList<Double> list = new ArrayList<>();
list.add(5.5); // 5.5 is automatically converted to new Double(5.5)
list.add(3.0); // 3.0 is automatically converted to new Double(3.0)
Double doubleObject = list.get(0); // No casting is needed
double d = list.get(1); // Automatically converted to double

Instead of using a generic type, you could simply make the type element Object, which can accommodate any object type. However, using a specific concrete type can improve software reliability and readability because certain errors can be detected at compile time rather than at runtime. As in the previous example, since stack1 is declared as a String type with the GenericStack class, only strings can be added to the stack. It would be a compile error if you attempt to add an integer to the stack1.

To create a stack of strings, you use **new GenericStack<String>()** or **new GenericStack<>().** This could mislead you into thinking that the constructor of GenericStack should be defined as:
public GenericStack<E>() {…} This is WRONG. It should be defined as: **public GenericStack() {…}**.
As another example, the following class defines an array-based generic Bag with a default constructor.

```
public class ArrayBag<E> {
    private E[] elements;
    private int num;
    public ArrayBag() {   //constructor
        elements = (E[]) new Object[MAXSIZE];
        num = 0;
   }
   ...
}
```

- **Generic Method**

A generic type can be defined **for a static method**. A generic method is written with the types of the parameters not fully specified. For example, the following code segment defines a generic method print to print an array of objects.

```
public class GenericMethodDemo {
    public static void main(String[] args ) {
        Integer[] integers = {1, 2, 3, 4, 5};
        String[] strings = {"London", "Paris", "New York", "Austin"};
        GenericMethodDemo.<Integer>print(integers);
        GenericMethodDemo.<String>print(strings);

    public static <E> void print(E[] list) {
        for (int i = 0; i < list.length; i++)
            System.out.print(list[i] + " ");
            System.out.println();
    }
}
```

To declare a generic method, you place the generic type <E> immediately after the keyword static in the method header. To invoke a generic method, prefix the method name with the actual type in angle brackets. More example:

```
static Integer middle(Integer [] data)
static Character middle(Character [] data)
```

The only difference of the above statements is the type. We can write a generic method with the types of the parameters not fully specified:

```
static <T> T middle(T [] data)
```

<T> is a generic type parameter appears right before the return type, and T can be any valid identifiers, but must be a class type. The following method signature uses 2 generic type parameters:

```
static <S, T> boolean most(S[] s, S skey, T[] t, T tkey)
```