Generics

- add stability to your code by making more of your bugs detectable at compile time,
- ability to create polymorphic algorithms

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

Generic Types

```java
public class GenericExample<T> {
    private T value;

    public GenericExample(T value){
        this.value = value;
    }

    public T getT(){
        return value;
    }

    public static void main(String[] args) {
        GenericExample<Integer> intObject = new GenericExample<>(1);
        Integer valueInteger = intObject.getT();

        GenericExample<String> stringObject = new GenericExample<>("word");
        String valueString = stringObject.getT();
    }
}
```

Generic Methods

```java
public class GenericExample {

    public static <T> T getTheFirst(List<T> list){
        return list.get(0);
    }

    public static void main(String[] args) {
        List<Integer> listOfInts = new ArrayList<>();

        listOfInts.add(0);
        Integer intValue = getTheFirst(listOfInts);

        List<String> listOfStrings = new ArrayList<>();

        listOfStrings.add(“Java is the best!”);
        String stringValue = getTheFirst(listOfStrings);
    }
}
```

Names of type parameters

- E: Element
- K: Key
- N: Number
- T: Type (generic)
- V: Value
- S, U, V, and so on: Second, third, and fourth types in a multiparameter situation

```
public class GenericExample<T, U, V> {
     private T valueT;
     private U valueU;
     private V valueV;

…
}
```

Wildcards

Wildcard (<?>) is specifies an unknown type using generic code.

```java
void printList(List<?> myList){
   // myList can be accessed
   // but you can't add to the list
}
```

Bounded Types

<? extends UpperBoundType>

<? super LowerBoundType>

Examples

<? extends Number>

<? super Set>