Agenda

- Generics
- Generic class
- Generic method
- ~~Generic Limitations~~
- ~~Generic Interfaces~~

Generic Programming

- Code is said to be generic if same code can be used for various (practically all) types.
- Best example:
  - Data structure e.g. Stack, Queue, Linked List, ...
  - Algorithms e.g. Sorting, Searching, ...
- Two ways to do Generic Programming in Java
  1. using java.lang.Object class -- Non typesafe
  2. using Generics -- Typesafe

1.Generics using Object class

```
class Box {
  private Object obj;
  public void set(Object obj) {
    this.obj = obj;
    }
  public Object get() {
    return this.obj;
    }
}
```

```
Box b1 = new Box();
b1.set("Sunbeam");
String obj1 = (String)b1.get();
System.out.println("obj1 : " + obj1);

Box b2 = new Box();
b2.set(new Date());
Date obj2 = (Date)b2.get();
System.out.println("obj2 : " + obj2);

Box b3 = new Box();
b3.set(new Integer(11));
String obj3 = (String)b3.get(); // ClassCastException
System.out.println("obj3 : " + obj3);
```

2.Generics using Generics

- Added in Java 5.0.
- Similar to templates in C++.
- We can implement
    1. Generic classes
    2. Generic methods
    3. Generic interfaces
- Advantages of Generics
    - Stronger type checking at compile time i.e. type-safe coding.
    - Explicit type casting is not required.
    - Generic data structure and algorithm implementation.

Generic classes

- Implementing a generic class

```
class Box<TYPE> {
  private TYPE obj;
  s
  public void set(TYPE obj) {
    this.obj = obj;
  }
  public TYPE get() {
    return this.obj;
  }
}
```

```
Box<String> b1 = new Box<String>();
b1.set("Sunbeam");
String obj1 = b1.get();
System.out.println("obj1 : " + obj1);

Box<Date> b2 = new Box<Date>();
b2.set(new Date());
Date obj2 = b2.get();
System.out.println("obj2 : " + obj2);

Box<Integer> b3 = new Box<Integer>();
b3.set(new Integer(11));
String obj3 = b3.get(); // Compiler Error
System.out.println("obj3 : " + obj3);
```


- Instantiating generic class

```
Box<String> b1 = new Box<String>(); // okay

Box<String> b2 = new Box<>(); // okay -- type inference
```

```
Box<> b3 = new Box<>(); // error -- type must be given while creating generic
class reference, as reference cannot be auto-detected

Box<Object> b4 = new Box<String>(); // error

Box b5 = new Box(); // okay -- internally considered Object type -- compiler
warning "raw types"

Box<Object> b6 = new Box<Object>(); // okay -- Not usually required/used
```

Generic types naming convention

1. T : Type
2. N : Number
3. E : Element
4. K : Key
5. V : Value
6. S,U,R : Additional type param

Bounded Generic types

- Bounded generic parameter restricts data type that can be used as type argument.
- Decided by the developer of the generic class.

```
class Box<T extends Number>{
    private T obj;

    public T getObj() {
        return obj;
    }

    public void setObj(T obj) {
        this.obj = obj;
    }
}
```

- The Box<> can now be used only for the classes inherited from the Number class.

```
Box<Number> b1 = new Box<>(); // okay
Box<Boolean> b2 = new Box<>(); // error
Box<Character> b3 = new Box<>(); // error
Box<String> b4 = new Box<>(); // error
Box<Integer> b5 = new Box<>(); // okay
Box<Double> b6 = new Box<>(); // okay
Box<Date> b7 = new Box<>(); // error
Box<Object> b8 = new Box<>(); // error
```

Unbounded Generic Types

- Unbounded generic type is indicated with wild-card "?".
- Can be given while declaring `generic class reference`.
- Remember unbounded work for class references and not for class types.

```
class Box<T> {
  private T obj;

  public Box(T obj) {
    this.obj = obj;
  }

  public T get() {
    return this.obj;
  }

  public void set(T obj) {
    this.obj = obj;
  }
}

public static void printBox(Box<?> b) {
  Object obj = b.get();
  System.out.println("Box contains: " + obj);
}

Box<String> sb = new Box<String>("DAC");
printBox(sb); // okay
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Date> db = new Box<Date>(new Date());
printBox(db); // okay
Box<Float> fb = new Box<Float>(200.5f);
printBox(fb); // okay
```

Upper bounded generic types

- Generic param type can be the given class or its sub-class.

```
public static void printBox(Box<? extends Number> b) {
Object obj = b.get();
System.out.println("Box contains: " + obj);
}

Box<String> sb = new Box<String>("DAC");
printBox(sb); // error
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Date> db = new Box<Date>(new Date());
```

```
printBox(db); // error
Box<Float> fb = new Box<Float>(200.5);
printBox(fb); // okay
```

- Here the upper bound is set (to Number) that means all the classes that inherits Number are allowed

Lower bounded generic types

- Generic param type can be the given class or its super-class.

```
public static void printBox(Box<? super Integer> b) {
Object obj = b.get();
System.out.println("Box contains: " + obj);
}

Box<String> sb = new Box<String>("DAC");
printBox(sb); // error
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Date> db = new Box<Date>(new Date());
printBox(db); // error
Box<Float> fb = new Box<Float>(200.5f);
printBox(fb); // error
Box<Number> nb = new Box<Number>(null);
printBox(nb); // okay
```

- Here the lower bound is set (to Integer) that means all the classes that are super classes of that lower bound class are allowed.

Generic Methods

- Generic methods are used to implement generic algorithms.
- Example

```
    // Not Type-safe
//  public static void printArray(Object[] arr) {
//      for (Object element : arr) {
//          System.out.println(element);
//      }
//  }

    // Type-safe
    public static <Type> void printArray(Type[] arr) {
        for (Type element : arr) {
            System.out.println(element);
        }
    }

    public static void main(String[] args) {
```

```
        String[] arr = { "Rohan", "Nilesh", "Amit" };
        printArray(arr);

        Integer[] arr2 = { 10, 20, 30, 40 };
        Program01.<Integer>printArray(arr2);

        Double[] arr3 = { 10.11, 20.12, 30.13 };
        // printArray(arr3); // type is inferred
        // Program01.<Integer>printArray(arr3); // compiler error
        Program01.<Double>printArray(arr3);// OK
}
```