Object Oriented Programming with Java

Sandeep Kulange

Agenda

- Boxing
- UnBoxing
- Generic programming using Object class
- Wrapper class hierarchy(Revision)
- Generic Programming Using Generics
- Why Generics
- Type Parameters
- Bounded Type Parameter
- Wild Card
- Generic Method
- Restrictions on generics.

Boxing

- Process of converting state of variable of primitive type into non primitive type is called as boxing.

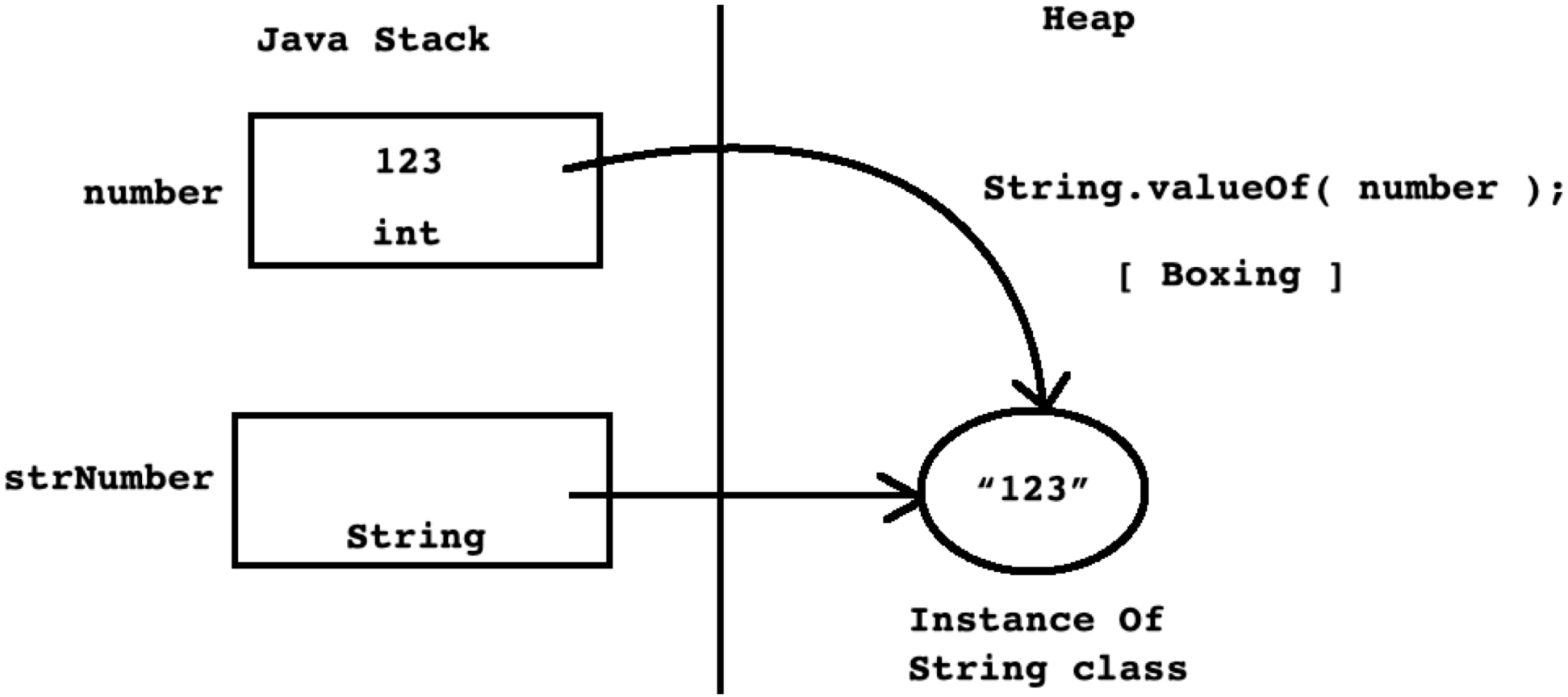
```
int number = 123;
String s1 = Integer.toString( number ); //Boxing
String s2 = String.valueOf( number );   //Boxing
```

- If boxing is done implicitly then it is called as auto-boxing.

```
int number = 123;
Object obj = number;
//Object obj = Integer.valueOf( number );
```

Boxing

```
int number = 123;
String strNumber = String.valueOf( number ); //Boxing
```



Unboxing

- Process of converting value of variable non primitive into primitive type is called as unboxing.

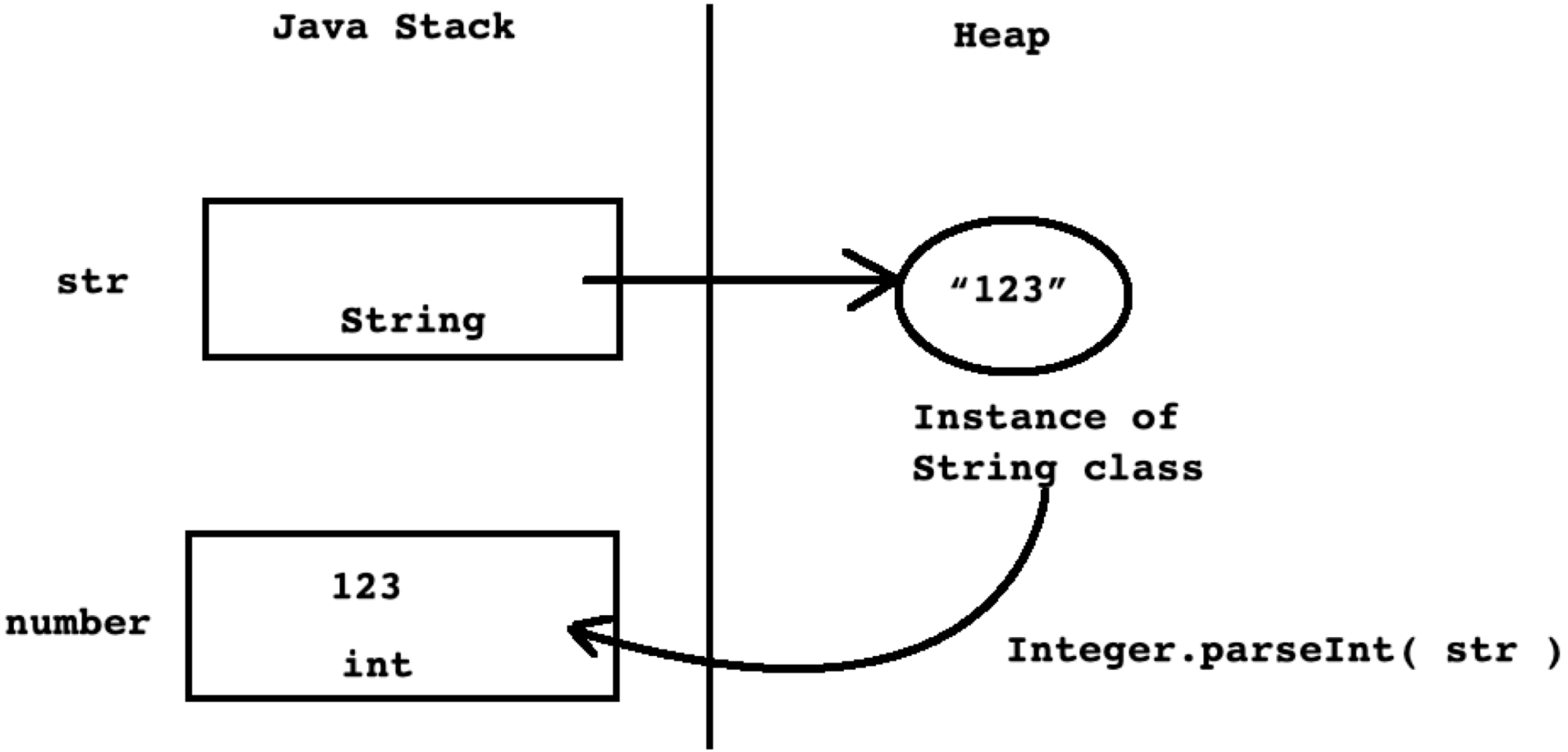
```
String str = "123";
int number = Integer.parseInt( str );    //UnBoxing
```

- If unboxing is done implicitly then it is called as auto-unboxing.

```
Integer n1 = new Integer("123");
int n2 = n1;
//int n2 = n1.intValue();
```

Unboxing

```
String str = “123”;
int number = Integer.parseInt( str ); //UnBoxing
```



Generic Programming

- If we want to write generic code in Java then we can use:
  1. java.lang.Object class
  2. Generics
- Generic Programming using java.lang.Object class.

```
class Box{
    private Object object;
    public Object getObject() {
        return object;
    }
    public void setObject(Object object) {
        this.object = object;
    }
}
```

Generic Programming(Using Object class)

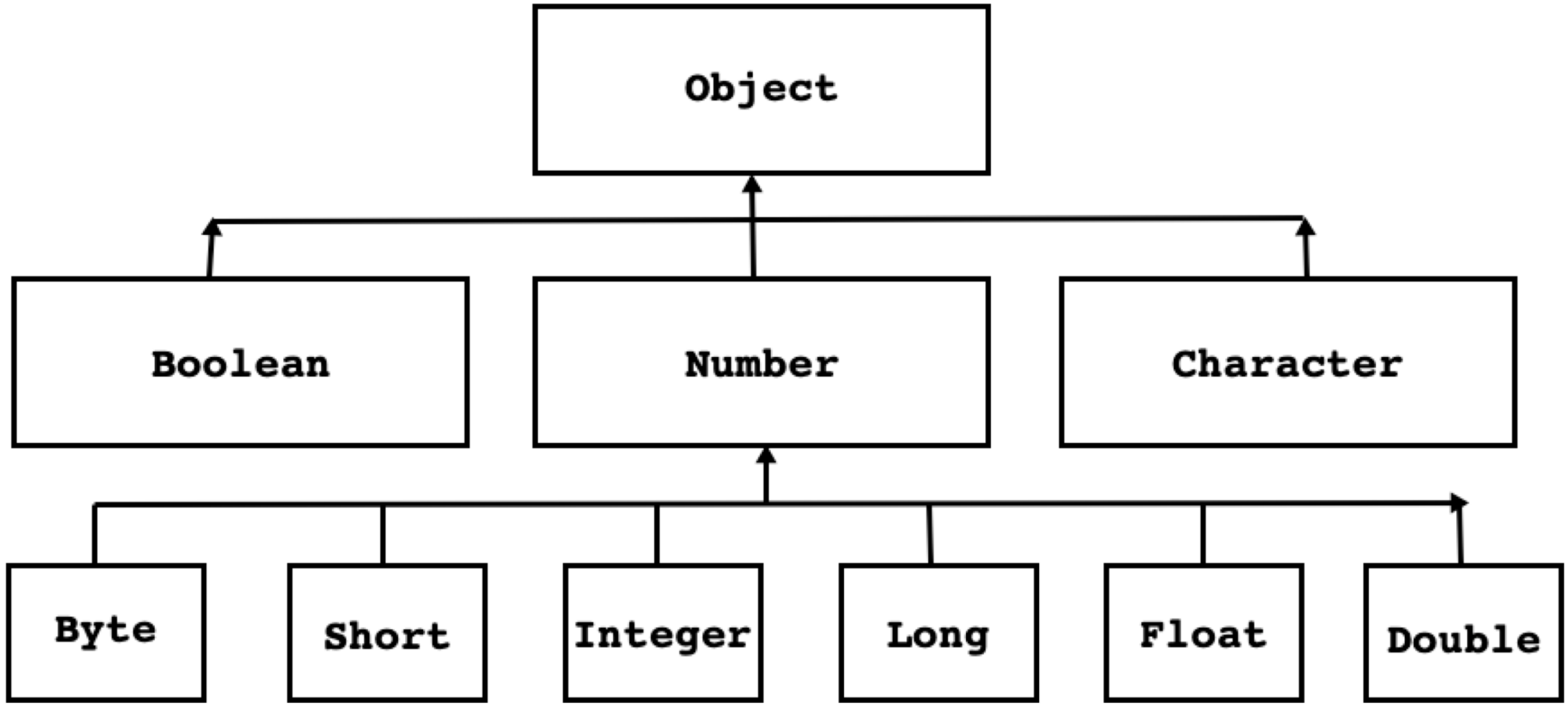
```java
public static void main(String[] args) {
    Box b1 = new Box( );
    b1.setObject(123);  //b1.setObject(Integer.valueOf(123));
    Integer n1 = (Integer) b1.getObject();  //Downcasting
    int n2 = n1.intValue();
}
```

```java
public static void main2(String[] args) {
    Box b1 = new Box( );
    b1.setObject( new Date());
    Date date = (Date) b1.getObject();  //Downcasting
    System.out.println(date.toString());
}
```

```java
public static void main(String[] args) {
    Box b1 = new Box( );
    b1.setObject( new Date() );
    String str =  (String) b1.getObject(); //Downcasting : ClassCastException
}
```

- Using java.lang.Object class, we can write generic code but we can not write type-safe generic code.
- If we want to write type safe generic code then we should use generics.

Wrapper Classes



Wrapper Classes

```
Number n1 = new Boolean( );      //Not OK

Number n2 = new Character( );    //Not OK

Number n3 = new Integer( ); //OK

Number n4 = new Double( );   //OK

Number n5 = new String( );   //Not OK
```

Generic Programming(Using Generics)

```
class Box<T>{    //T : Type Parameter
    private T object;
    public T getObject() {
        return object;
    }
    public void setObject(T object) {
        this.object = object;
    }
}
```

```
Box<Date> b1 = new Box<Date>( );    //Date : Type Argument

Box<Date> b1 = new Box<>( ); //Type Inference
```

- By passing, data type as a argument, we can write generic code in Java. Hence parameterized class/type is called as generics.
- If we specify type argument during declaration of reference variable then specifying type argument during instance creation is optional. It is also called as type inference.

Generic Programming(Using Generics)

```
1. Box<String> b1 = new Box<String>( ); //OK

2. Box<String> b2 = new Box<>( ); //OK

3. Box<Object> b3 = new Box<String>( ); //NOT OK

4. Box<Object> b4 = new Box<Object>( ); //OK
```

Generic Programming(Using Generics)

- If we instantiate parameterized type without type argument then java.lang.Object is considered as default type argument.

```
Box b1 = new Box( );     //Class Box : Raw Type
//Box<Object> b1 = new Box<>( );
```

- If we instantiate parameterized type without type argument then parameterized type is called raw type.

- During the instantiation of parameterized class, type argument must be reference type.

```
//Box<int> b1 = new Box< >( );   //Not OK
Box<Integer> b1 = new Box< >( );     //OK
```

Why Generics?

1. It gives us stronger type checking at compile time. In other words, we can write type safe code.
2. No need of explicit type casting.
3. We can implement generic data structure and algorithm.

Type Parameters Used In Java API

1. T : Type
2. N : Number
3. E : Element
4. K : Key
5. V : Value
6. S, U : Second Type Parameter Names

Bounded Type Parameter

- If we want to put restriction on data type that can be used as type argument then we should specify bounded type parameter.

```
class Box<T extends Number>{     //T is bounded type parameter
    private T object;
    public T getObject() {
        return object;
    }
    public void setObject(T object) {
        this.object = object;
    }
}
```

- Specifying bounded type parameter is a job of class implementor.

Bounded Type Parameter

```
1. Box<Number> b1 = new Box<>();     //OK

2. Box<Boolean> b2 = new Box<>();    //Not OK

3. Box<Character> b3 = new Box<>(); //Not OK

4. Box<String> b4 = new Box<>();     //Not OK

5. Box<Integer> b5 = new Box<>();    //OK

6. Box<Double> b6 = new Box<>();     //OK

7. Box<Date> b7 = new Box<>();       //Not OK
```

Wild Card

- In generics "?" is called as wild card which represents unknown type.

- **Types of wild card:**
  1. Unbounded wild card
  2. Upper bounded wild card
  3. Lower bounded wilds card

Unbounded Wild Card

```
private static void printRecord(ArrayList<?> list) {
    for (Object element : list) {
        System.out.println(element);
    }
}
```

- In above code, list will contain reference of ArrayList which can contain any type of element.

```
public static void main(String[] args) {
    ArrayList<Integer> integerList = Program.getIntegerList();
    Program.printRecord( integerList ); //OK

    ArrayList<Double> doubleList = Program.getDoubleList();
    Program.printRecord( doubleList );  //OK

    ArrayList<String> stringList = Program.getStringList();
    Program.printRecord( stringList );  //OK
}
```

Upper Bounded Wild Card

```
private static void printRecord(ArrayList<? extends Number > list) {
    for (Object element : list) {
        System.out.println(element);
    }
}
```

- In above code, list will contain reference of ArrayList which can contain Number and its sub type of elements.

```
public static void main(String[] args) {
    ArrayList<Integer> integerList = Program.getIntegerList();
    Program.printRecord( integerList ); //OK

    ArrayList<Double> doubleList = Program.getDoubleList();
    Program.printRecord( doubleList );  //OK

    ArrayList<String> stringList = Program.getStringList();
    Program.printRecord( stringList );  //Not OK
}
```

Lower Bounded Wild Card

```
private static void printRecord(ArrayList< ? super Integer > list) {
    for (Object element : list) {
        System.out.println(element);
    }
}
```

- In above code, list will contain reference of ArrayList which can contain Integer and its super type of elements.

```
public static void main(String[] args) {
    ArrayList<Integer> integerList = Program.getIntegerList();
    Program.printRecord( integerList ); //OK

    ArrayList<Double> doubleList = Program.getDoubleList();
    Program.printRecord( doubleList );  //NOT OK

    ArrayList<String> stringList = Program.getStringList();
    Program.printRecord( stringList );  //NOT OK
}
```

Generic Method

```
private static <T> void print( T obj ) {
    System.out.println(obj);
}

private static <T extends Number> void print( T obj ) {
    System.out.println(obj);
}
```

Restrictions on Generics

1. Cannot Instantiate Generic Types with Primitive Types

2. Cannot Create Instances of Type Parameters

3. Cannot Declare Static Fields Whose Types are Type Parameters

4. Cannot Use Casts or instanceof with Parameterized Types

5. Cannot Create Arrays of Parameterized Types

6. Cannot Create, Catch, or Throw Objects of Parameterized Types

7. Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Thank You.

[sandeepkulange@sunbeaminfo.com]