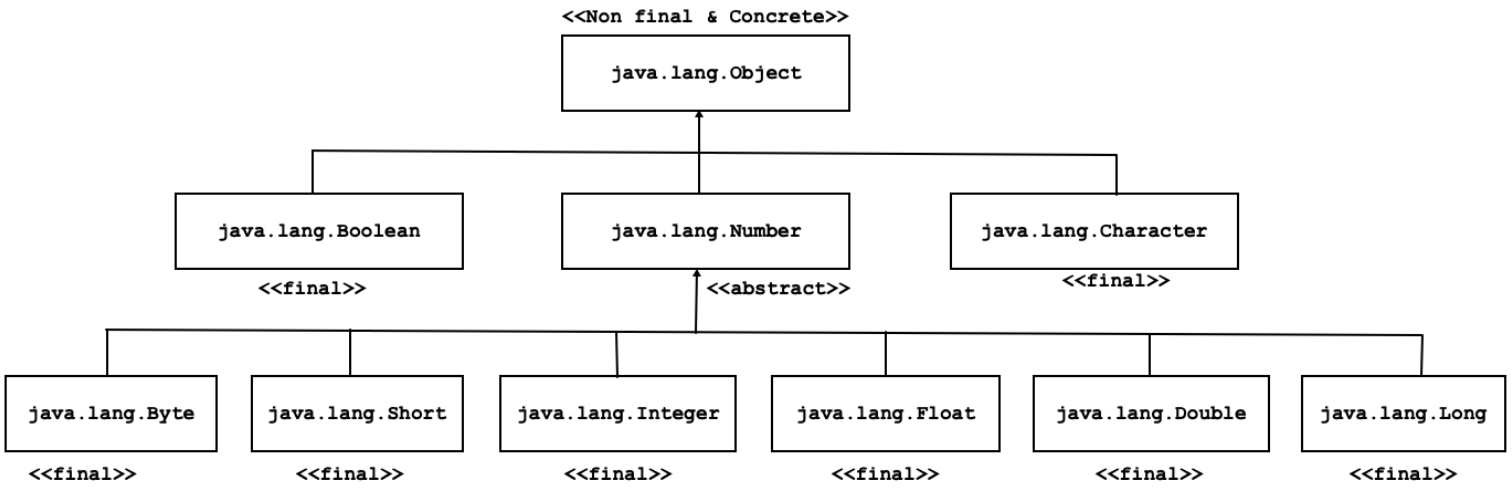
Generics

Revision

- Consider Wrapper class hierarchy:



- **Widening** is the process of converting value of variable of narrower type into wider type.

```
int num1 = 125;
double num2 = ( double )num1; //OK: Widening
double num3 = num1; //OK: Widening
```

- **Narrowing** is the process of converting value of variable of wider type into narrower type.

```
double num1 = 10.5;
int num2 = ( int )num1; //OK: Narrowing
int num3 = num1; //Not OK: Narrowing
```

- **Boxing** is the process of converting value of variable of primitive type into non primitive type. Consider below code:

```
int number = 125;
String str = String.valueOf( number );  //Boxing
Integer i = Integer.valueOf( number );  //Boxing
```

- If Boxing is done implicitly then it is called as **auto-boxing**. Consider below code:

```
int number = 125;
Object obj = number;  //Auto-Boxing
```

  - How it is treated implicitly?

```
int number = 125;
//First number is boxed into Integer instance
Integer i = Integer.valueOf( number );  //Boxing
//Then reference of Integer is converted into Object( Upcasting )
Object obj = i; //Upcasting
```

- **UnBoxing** is the process of converting value of variable of non primitive type into primitive type. Consider below code:

```
String str = "125";
int number = Integer.parseInt( str ); //UnBoxing
```

- If UnBoxing is done implicitly then it is called as **auto-unboxing**.Consider below code:

```
Integer i = new Integer( 125 );
int number = i; //Auto-Unboxing
```

- How it is working implicitly?

```
Integer i = new Integer( 125 );
int temp = i.intValue( ); //Unboxing
int number = temp;
```

- **Upcasting** is the process of converting reference of sub class into reference of super class. Consider below code:

```
Employee emp = new Employee("Sandeep", 10003778, 45000.50f );
Person p1 = ( Person )emp; //OK: Upcasting
Person p2 = emp; //OK: Upcasting

Person p3 = new Employee("Sandeep", 10003778, 45000.50f );  //OK: Upcasting
```

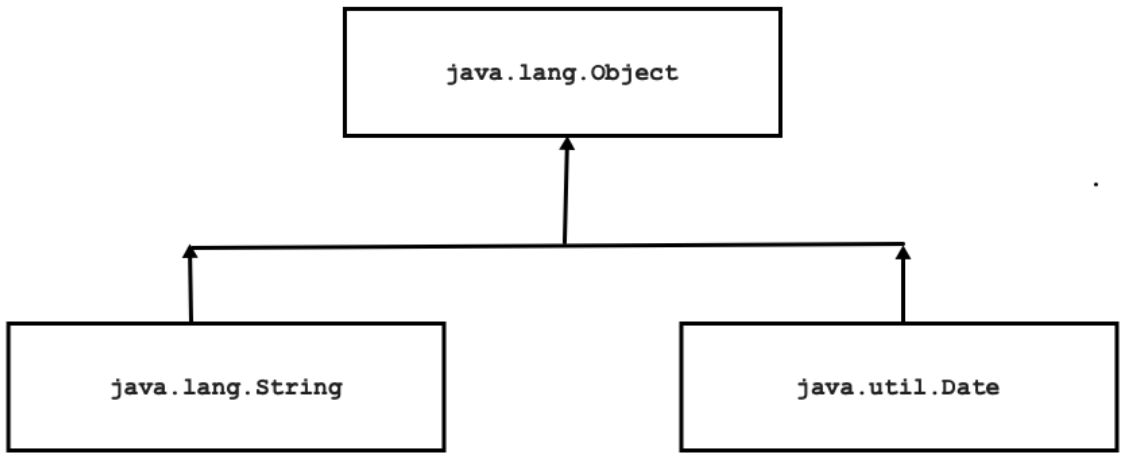
- **Downcasting** is the process of converting reference of super class into reference of sub class. Consider below code:

```
Person p = new Employee("Sandeep", 10003778, 45000.50f );  //OK: Upcasting
Employee emp = ( Employee )p; //OK: Downcasting
```

- If downcasting fails then JVM throws ClassCastException. Consider below code:

```
Person p = new Person("Sandeep", 40 );  //OK
Employee emp = ( Employee )p; //OK: Downcasting: ClassCastException
```

- Consider upcasting, downcasting and ClassCastException using below code:



  - Code Snippet 1:

```
Object o = new String("Sandeep"); //Upcasting
String str = ( String )o; //Downcasting: OK
```

  - Code Snippet 2:

```
Object o = new Date( ); //Upcasting
Date date = ( Date )o;  //Downcasting: OK
```

  - Code Snippet 3:

```
Object o = new Date( ); //Upcasting
//Downcasting should be done in Date but I am doing it in String
String str = ( String)o;  //Downcasting: ClassCastException
```

    - There is is-a relationship exist between String and Date. Hence during downcasting jvm is throwing ClassCastException.

Generic programming in Java

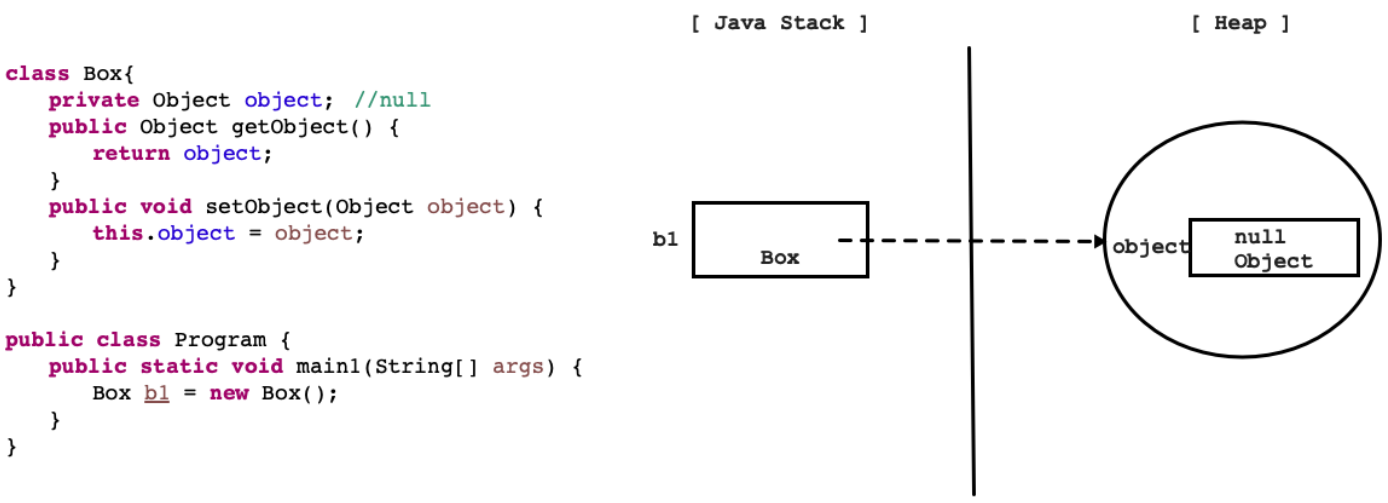
- In Java, we can write generic Code using 2 ways:
  - Using java.lang.Object class
  - Using Generics

- Consider Generic code implementation using java.lang.Object class:

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

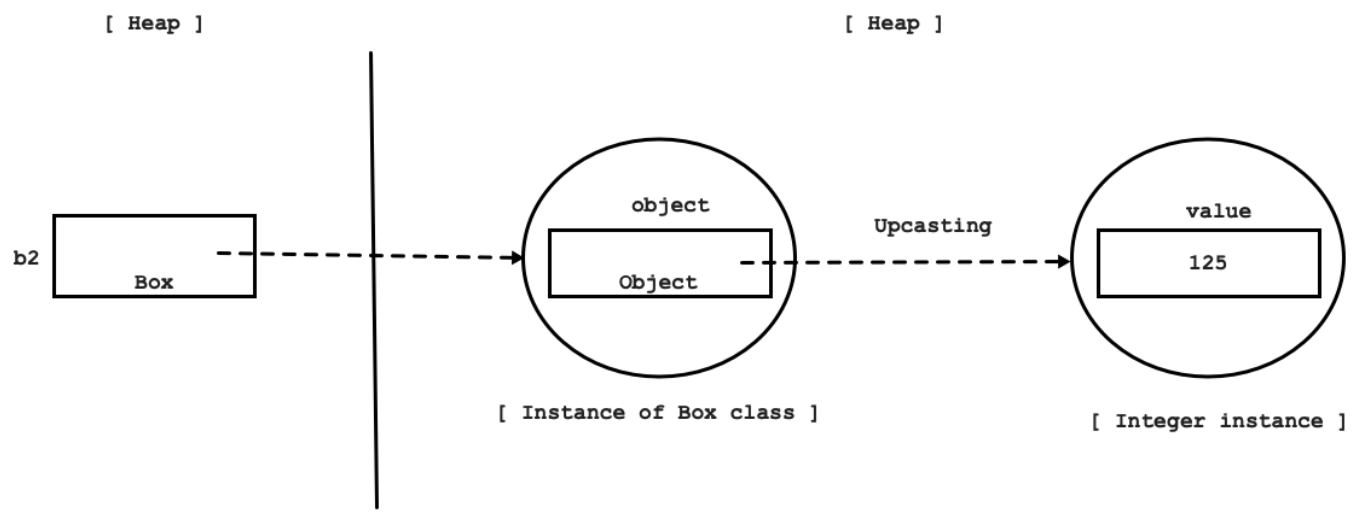
  - Code snippet 1:

```
Box b1 = new Box( );
```



  - Code snippet 2:

```
Box b2 = new Box( );
int num1 = 125;
b2.setObject( num1 ); //b2.setObject( Integer.valueOf( num1 ) ); //Auto-Boxing
```

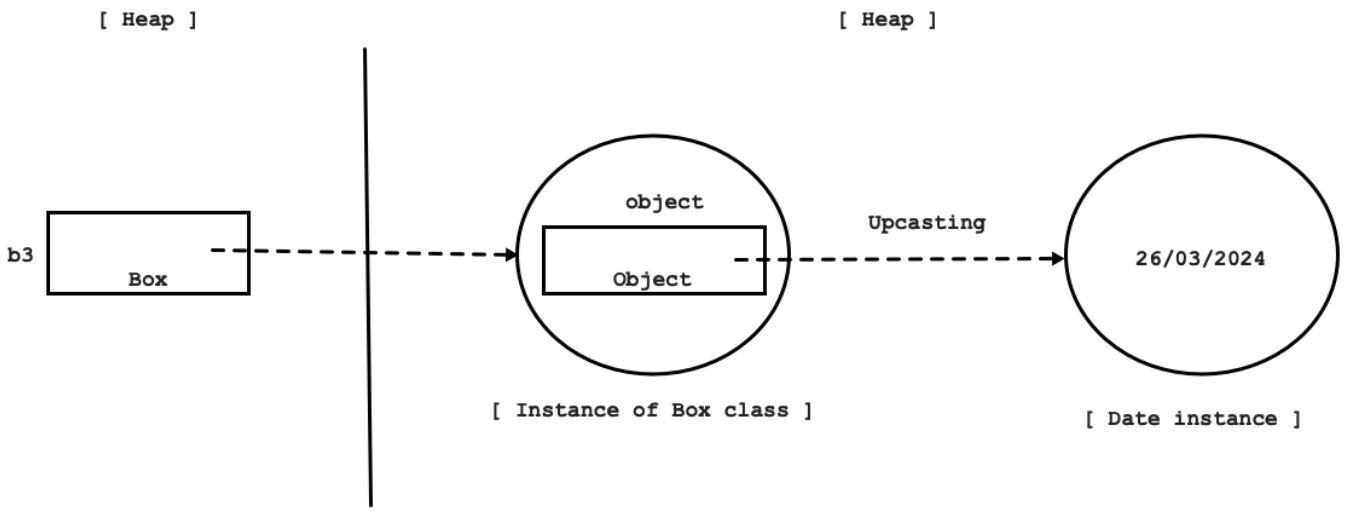


  - How will you retrieve int value? Consider code snippet 3:

```
Box b2 = new Box( );
int num1 = 125;
b2.setObject( num1 );
int num2 = ( int )b2.getObject( ); //Auto-UnBoxing
```

  - Code snippet 4:

```
Box b3 = new Box( );
Date dt1 = new Date( );
b3.setObject( dt1 ) //Object object = dt1; //Upcasting
```



- How will you retrieve date instance? Consider code snippet 5:

```
Box b3 = new Box( );
Date dt1 = new Date( );
b3.setObject( dt1 ) //Object object = dt1; //Upcasting
Date dt2 = ( Date )b3.getObject( );  //Downcasting
```

- Consider below code:

```
Box b = new Box( );
Date dt1 = new Date( );
b.setObject( dt1 ) //Object object = dt1; //Upcasting
String str = ( String )b.getObject( );  //Downcasting: ClassCastException
```

- Using java.lang.Object class, we can write generic code but we can not define type-safe generic code. Hence to write type-safe generic code, we should use generics in Java.
- Consider generic code implementation using generics:

```
//Now Box is parameterized type
class Box<T>{ //T is called as Type Parameter name
  private T object;
  public T getObject() {
    return object;
  }
  public void setObject(T object) {
    this.object = object;
  }
}

public class Program {
  public static void main(String[] args) {
    Box<Date> box = new Box<Date>();  //Date: Type Argument

    box.setObject( new Date() );

    Date date = box.getObject();

    System.out.println( date );

  }
}
```

- **Why Use Generics?**
  - Generics gives us stronger type checking at compile time.
  - It eliminates need of explicit need of type casting.
  - It helps developer to define generic data structure and generic algorithm.
  - Reference: https://docs.oracle.com/javase/tutorial/java/generics/why.html
- **Type argument syntax**
  - Code snippet 1:

```
Box<Date> box = new Box<Date>();  //OK
```

- Code snippet 2:

```
Box<Date> box = new Box< >();  //OK: Type Inference
```

- Code snippet 3:

```
Box<> box = new Box<Date>();  //Not OK
```

- Code snippet 4:

```
Box<> box = new Box<>();  //Not OK
```

- Code snippet 5:

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

- Code snippet 6:

```
Box<Integer> box = new Box<>();   //OK
```

- Code snippet 7:

```
Box<Number> box = new Box<Integer>();  //Not OK
```

- Code snippet 8:

```
Box<Integer> box = new Box<Number>(); //Not OK
```

- Code snippet 9:

```
Box box = new Box();  //OK: Here Box is raw type
//Box<Object> box = new Box<>();  //Same as above code
```

- Code snippet 10:

```
List<Integer> list = new ArrayList<Integer>( ); //OK
List<Number> list = new ArrayList<Integer>( ); //Not OK:
```

- **Commonly used type parameter names:**
  - In Java generics, type parameters are placeholders for actual types. Below are the commonly used type parameter names:
    - **T** : Type
    - **E** : Element
    - **N** : Number
    - **K** : Key
    - **V** : Value
    - **R** : Return Type
    - **S,U,V** : When more than one type parameter is needed we can use it.
- Consider below **code with more than one type parameter name**:

```
interface Map<K, V>{
  K getKey( );
```

```
  V getValue( );
}
```

```
class HashTable< K, V > implements Map<K, V>{
  private K key;
  private V value;

  public HashTable(K key, V value) {
    this.key = key;
    this.value = value;
  }
  @Override
  public K getKey() {
    return this.key;
  }
  @Override
  public V getValue() {
    return this.value;
  }
}
```

```
public class Program {
  public static void main(String[] args) {
    HashTable<Integer, String> ht = new HashTable<>(1, "CDAC");

    Integer key = ht.getKey();

    String value = ht.getValue();
  }
}
```

- **Bounded Type Parameter**
  - Using bounded type parameter, we can put restriction on data type that can be used as type argument. In below code, we are forcing to specify Number or its sub types only:

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

```
public class Program {
  public static void main(String[] args) {
    Box<Number> b1 = new Box<>(); //OK

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

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

    Box<Integer> b4 = new Box<>();  //OK

    Box<Double> b5 = new Box<>(); //OK

    //Box<String> b6 = new Box<>();   //Not OK

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

- **java.util.ArrayList** is a resizable array.
  - Consider ArrayList of Object

```
public static  ArrayList<Object> getObjectList( ) {
  ArrayList<Object> list = new ArrayList<>( );
  list.add(true);
  list.add('A');
  list.add(123);
  list.add(3.14);
  list.add("Sandeep");
  list.add(new Date());
  return list;
}
```

- Consider ArrayList of Number

```
public static  ArrayList<Number> getNumberList( ) {
  ArrayList<Number> list = new ArrayList<>( );
  list.add( 10 );     //list.add( Integer.valueOf( 10 ) );
  list.add( 20.1f );  //list.add( Float.valueOf( 20 ) );
  list.add( 30.2d );  //list.add( Double.valueOf( 30 ) );
  return list;
}
```

- Consider ArrayList of Integer

```
public static  ArrayList<Integer> getIntegerList( ) {
  ArrayList<Integer> list = new ArrayList<>( );
  list.add( 10 ); //list.add( Integer.valueOf( 10 ) );
  list.add( 20 ); //list.add( Integer.valueOf( 20 ) );
  list.add( 30 ); //list.add( Integer.valueOf( 30 ) );
  return list;
}
```

- Consider ArrayList of Double

```
public static  ArrayList<Double> getDoubleList( ) {
  ArrayList<Double> list = new ArrayList<>( );
  list.add( 10.1 );   //list.add( Double.valueOf( 10.1 ) );
  list.add( 20.2 );   //list.add( Double.valueOf( 20.2 ) );
  list.add( 30.3 );   //list.add( Double.valueOf( 30.3 ) );
  return list;
}
```

- Consider ArrayList of String

```
public static  ArrayList<String> getStringList( ) {
  ArrayList<String> list = new ArrayList<>( );
  list.add( "Sandeep" );
  list.add( "Prathamesh" );
  list.add( "Soham" );
  return list;
}
```

- **Wild Card**
  - In Generics **?** is called as wild card which represents unknown type. Below are the types of wild card:
    - UnBounded Wild Card
    - Upper Bounded Wild Card
    - Lower Bounded Wild Card
- **UnBounded Wild Card**
  - Consider below code:

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

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

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

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

- **Upper Bounded Wild Card**
  - Consider below code:

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

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

  ArrayList<Number> numberList = Program.getNumberList();
  Program.printList( numberList  );   //OK

  ArrayList<Integer> integerList = Program.getIntegerList();
  Program.printList( integerList  );  //OK

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

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

- **Lower Bounded Wild Card**
  - Consider below code:

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

```
public static void main(String[] args) {
  ArrayList<Object> objectList = Program.getObjectList();
  Program.printList( objectList  );   //OK

  ArrayList<Number> numberList = Program.getNumberList();
  Program.printList( numberList  );   //OK

  ArrayList<Integer> integerList = Program.getIntegerList();
  Program.printList( integerList  );  //OK

  ArrayList<Double> doubleList = Program.getDoubleList();
  //Program.printList( doubleList  ); //Not OK

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

Restrictions on Generics

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters

- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or instanceof with Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type
- **Reference:** https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html