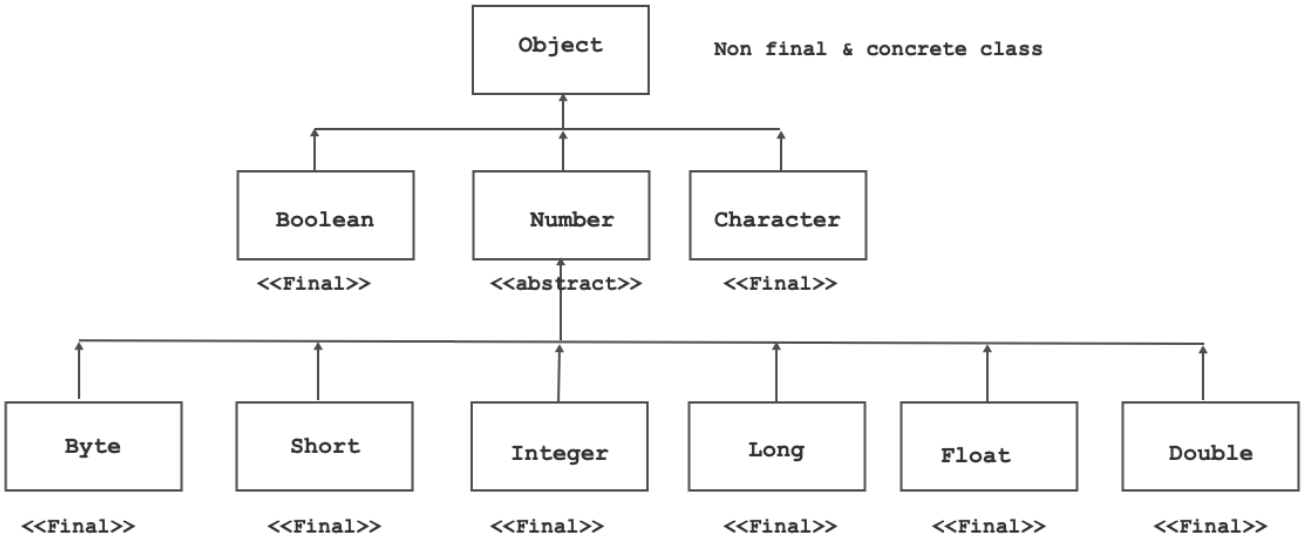
Day 15

Wrapper class



- Primtive Types in Java
  - boolean, byte, char, short, int, float, double, long.
- In Java, primitive types are not classes. If we want to process values of primitive type then we can use corresponding class. It is called wrapper class.
- All the wrapper classes are final and declared in java.lang package.
- All the wrapper classes implments java.lang.Comparable and java.io.Serializable interface.
- Any Wrapper class do not contain parameterless constructor
- Super class reference variable can contain refrence of instance of sub class. It is called as upcasting.

```
Object o1 = new Boolean( true );   //OK: Upcasting
Object o2 = new Character( 'A' ); //OK: Upcasting
Object o3 = new Byte( 123 );       //OK: Upcasting
Number n1 = new Byte( 123 );       //OK: Upcasting
Byte b1 = new Byte( 123 );         //OK
Object o4 = new Short( 9999 );       //OK: Upcasting
Number n2 = new Short(9999 );       //OK: Upcasting
Object o5 = new Integer( 123456 );       //OK: Upcasting
Number n3 = new Integer( 123456 );       //OK: Upcasting
Object o6 = new Long(123456789 );       //OK: Upcasting
Number n4 = new Long(123456789);       //OK: Upcasting
Object o7 = new Float(3.14f);       //OK: Upcasting
Number n5 = new Float(3.14f);       //OK: Upcasting
Object o8 = new Double(3.142d);       //OK: Upcasting
Number n6 = new Double(3.142d);       //OK: Upcasting
```

- Consider example of integer

```
int n1 = new int( 10 ); //Not OK
Integer n1 = new Integer( 10 ); //OK
Number n2 = new Integer( 10 ); //OK: Upcasting
Object n3 = new Integer( 10 ); //OK: Upcasting
```

Boxing and Auto-Boxing

- Consider following example:

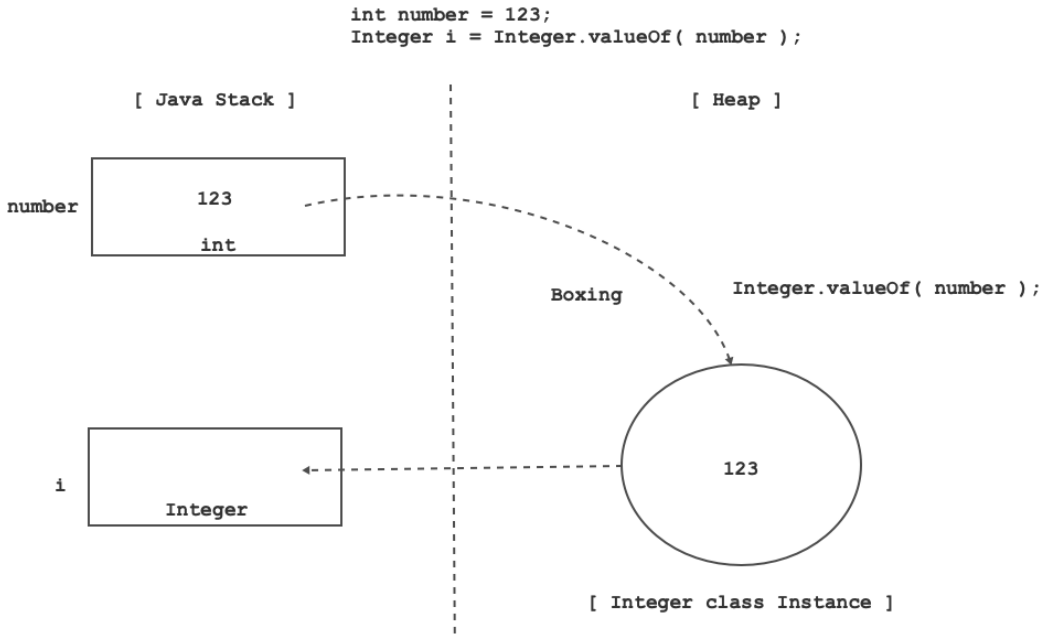
```
int number = 123;
Integer i1 = new Integer( number ); //Boxing
```

```
int number = 123;
Integer i2 = Integet.valueOf( number ); //Boxing
```

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

```
int number = 123;
String s2 = String.valueOf( number );
```

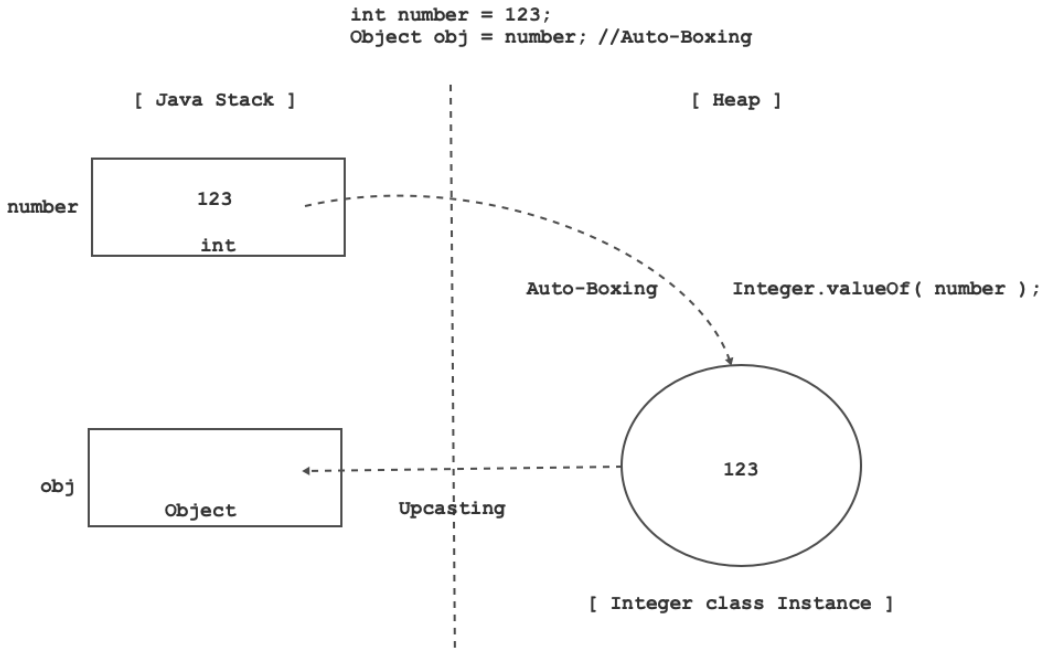
- Process of converting value of variable of primitive type into non primitive type is called as boxing.



- Consider following example:

```
public static void main(String[] args) {
    int number = 123;
    Object obj = number;    //Auto-Boxing
    //Integer i = Integer.valueOf(number)   //Boxing
    //Object o = i; //Upcasting
    System.out.println( obj );
}
```

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



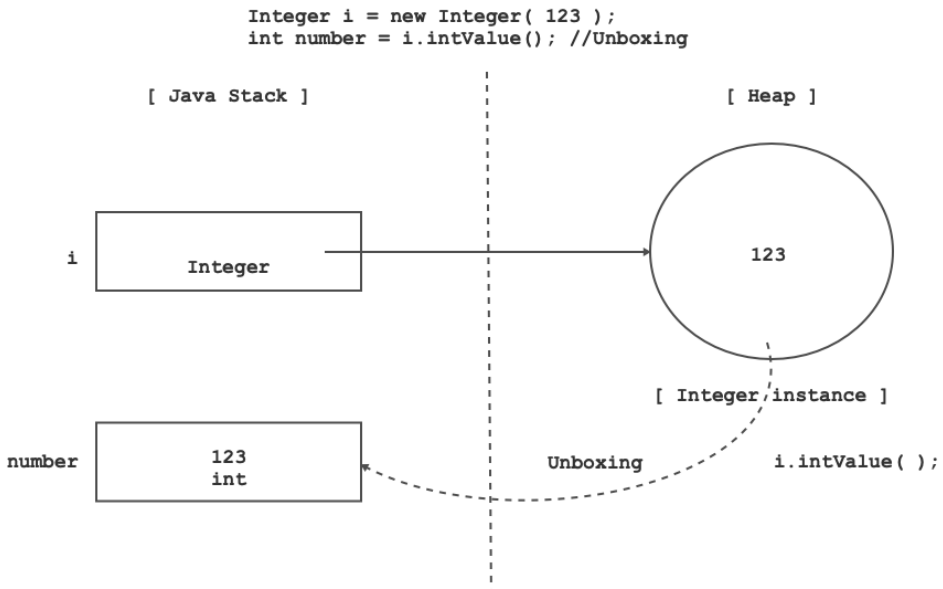
UnBoxing and Auto-Unboxing

- Consider following example:

```
//Integer i = new Integer( "123" );
Integer i = new Integer( 123 );
int number = i.intValue( ); //UnBoxing
```

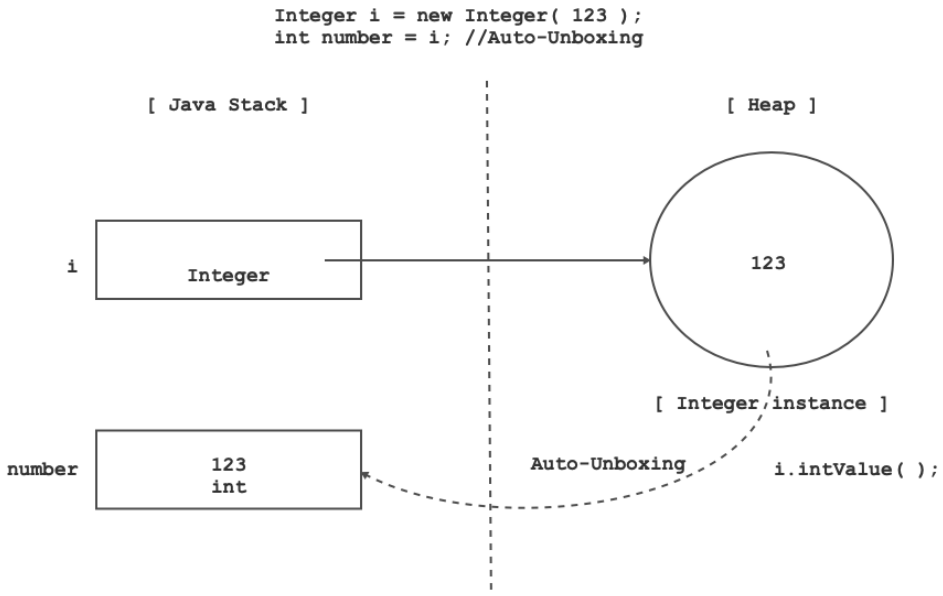
```
String str = "456";
int number = Integer.parseInt( str ); //UnBoxing
```

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



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

```
    public static void main(String[] args) {
    Integer i = new Integer(123);
    int number = i; //Auto-Unboxing
    //int number = i.intValue();
    System.out.println("Number  :   "+number);
}
```



Upcasting and downcasting

- Upcasting definition
  - We can convert reference of sub class into reference of super class. It is called as upcasting.

```

```

```
Employee emp = new Employee();
//Person p = (Person)emp; //OK: Upcasting
Person p = emp; //OK: Upcasting
```

- We can directly store reference of sub class instance into super class reference variable. It is called as upcasting.

```
Person p = new Employee();  //Upcasting
```

- If we want to reduce object/instance dependency in the code then we should use upcasting.
- In case of upcasting, using super class reference variable we can access:
  - non private fields of super class.
  - non private methods of super class.
  - overriden methods of sub class.
- Downcasting definition
  - We can convert reference super class into reference of sub class. It is called as downcasting.

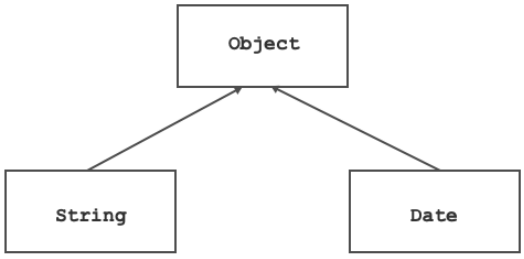
```
Person p = new Employee( ); //Upcasting
Employee emp = (Employee)p; //Downcasting
```

- If JVM failed to do downcasting then it throws ClassCastException.

```
  Employee e = new Person();  //Compile time checking =>  Compiler
Error

  Person p = new Person( );
  Employee emp = (Employee)p; //Downcasting: ClassCastException
```

- In case of upcasting, using super class reference variable, we can not access non private fields and non overriden methods of sub class. If we want to access it then we should do downcasting.




```
String str = new String("CDAC");//OK

Object o1 = new String("CDAC"); //OK: Upcasting
String s1 = (String)o1;         //OK: Downcasting


Object o2 = null;
String s2 = (String)o2; //OK: Downcasting
```

```
Date date = new Date( );//OK

Object o1 = new Date( );  //OK: Upcasting
Date dt1 = (Date)o1;      //Ok: Downcasting


Object o2 = null;
Date dt2 = (Date)o2; //OK: Downcasting
```

```
Object o3 = new Date();   //Ok: Upcasting
Date dt3 = (Date)o3;      //OK: Downcasting
String s3 = (String)o3;   //ClassCastException
```

- Consider following code:

```
class A{
  public void print( ){
    System.out.println("A.print");
  }
  public void display( ){
    System.out.println("A.display");
  }
}
class B extends A{
  @Override
  public void print( ){
    System.out.println("B.print");
  }
  public void show( ){
    System.out.println("B.show");
  }
}
```

```
class Program{
  public static void main( String[] args ){
    A a1 = new A(); //OK
    a1.print(); //A.print

    B b1 = new B( );  //OK
    b1.print(); //B.print => Due to shadowing preference will be given
to B.print

    A a2 = new B(); //OK: Upcasting
    a2.print( );  //B.print
```

```
    a2.display( );//A.display
    a2.show( ); //Compiler error

    B b2 = new A(); //Compiler error
    b2.print( );
  }
}
```

Generic Programming

- Let us define stack to store boolean elements

```
  class Stack{
    private int top = -1;
    private boolean[] arr;
    public Stack(  ) {
      this( 5 );
    }
    public Stack( int size ) {
      this.arr = new boolean[ size ];
    }
    public boolean empty( ) {
      return this.top == -1;
    }
    public boolean full( ) {
      return this.top == this.arr.length - 1;
    }
    public void push( boolean element ) throws StackOverflowException
{
      if( this.full())
        throw new StackOverflowException("Stack is full");
      this.top = this.top + 1;
      this.arr[ this.top ] = element;
    }
    public boolean peek( ) throws StackUnderflowException {
      if( this.empty())
        throw new StackUnderflowException("Stack is empty");
      return this.arr[ this.top ];
    }
    public void pop( ) throws StackUnderflowException {
      if( this.empty())
        throw new StackUnderflowException("Stack is empty");
      this.top = this.top - 1;
    }
  }
```

- Let us define stack to store integer elements

```
```

```
class Stack{
  private int top = -1;
  private int[] arr;
  public Stack(  ) {
    this( 5 );
  }
  public Stack( int size ) {
    this.arr = new int[ size ];
  }
  public boolean empty( ) {
    return this.top == -1;
  }
  public boolean full( ) {
    return this.top == this.arr.length - 1;
  }
  public void push( int element ) throws StackOverflowException {
    if( this.full())
      throw new StackOverflowException("Stack is full");
    this.top = this.top + 1;
    this.arr[ this.top ] = element;
  }
  public int peek( ) throws StackUnderflowException {
    if( this.empty())
      throw new StackUnderflowException("Stack is empty");
    return this.arr[ this.top ];
  }
  public void pop( ) throws StackUnderflowException {
    if( this.empty())
      throw new StackUnderflowException("Stack is empty");
    this.top = this.top - 1;
  }
}
```

- Let us define stack to store double elements

```
class Stack{
  private int top = -1;
  private double[] arr;
  public Stack(  ) {
    this( 5 );
  }
  public Stack( int size ) {
    this.arr = new double[ size ];
  }
  public boolean empty( ) {
    return this.top == -1;
  }
  public boolean full( ) {
    return this.top == this.arr.length - 1;
  }
  public void push( double element ) throws StackOverflowException {
```

```
    if( this.full())
      throw new StackOverflowException("Stack is full");
    this.top = this.top + 1;
    this.arr[ this.top ] = element;
  }
  public double peek( ) throws StackUnderflowException {
    if( this.empty())
      throw new StackUnderflowException("Stack is empty");
    return this.arr[ this.top ];
  }
  public void pop( ) throws StackUnderflowException {
    if( this.empty())
      throw new StackUnderflowException("Stack is empty");
    this.top = this.top - 1;
  }
}
```

- If we want to reduce developers effort then we should do generic programming.
- In Java, we can write generic code using:
  1. java.lang.Object class
  2. Generics

Generic programming using java.lang.Object class

- We can use java.lang.Object class to store value of primitive as well as non-primitive. Hence to write generic code we use java.lang.Object class.

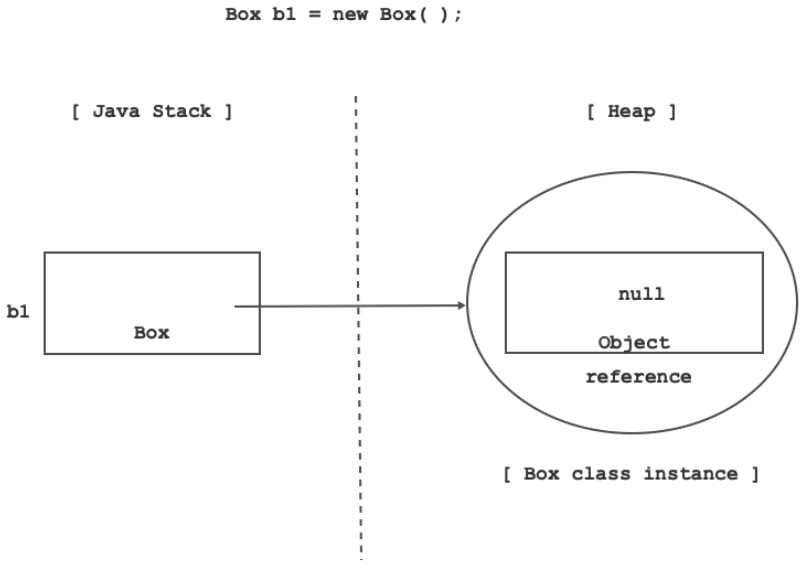
```
int number = 123;
Object obj = number; //OK
/*
Integer i = Integer.valueOf( number ); //Boxing
Object obj = i; //Upcasting
*/
```

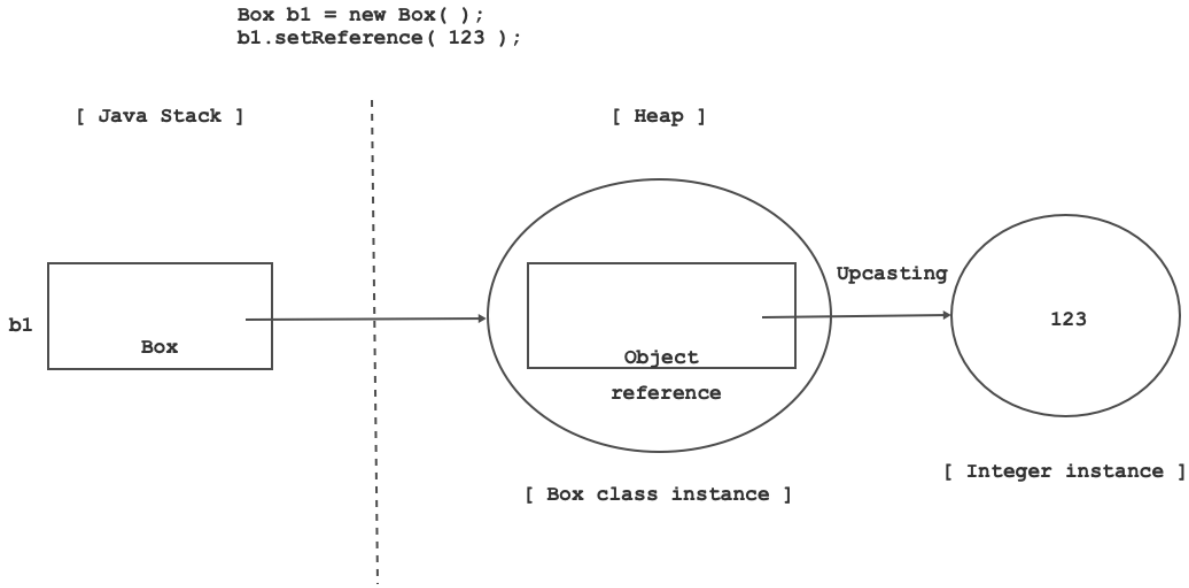
```
Object obj = new Date();  //OK
```

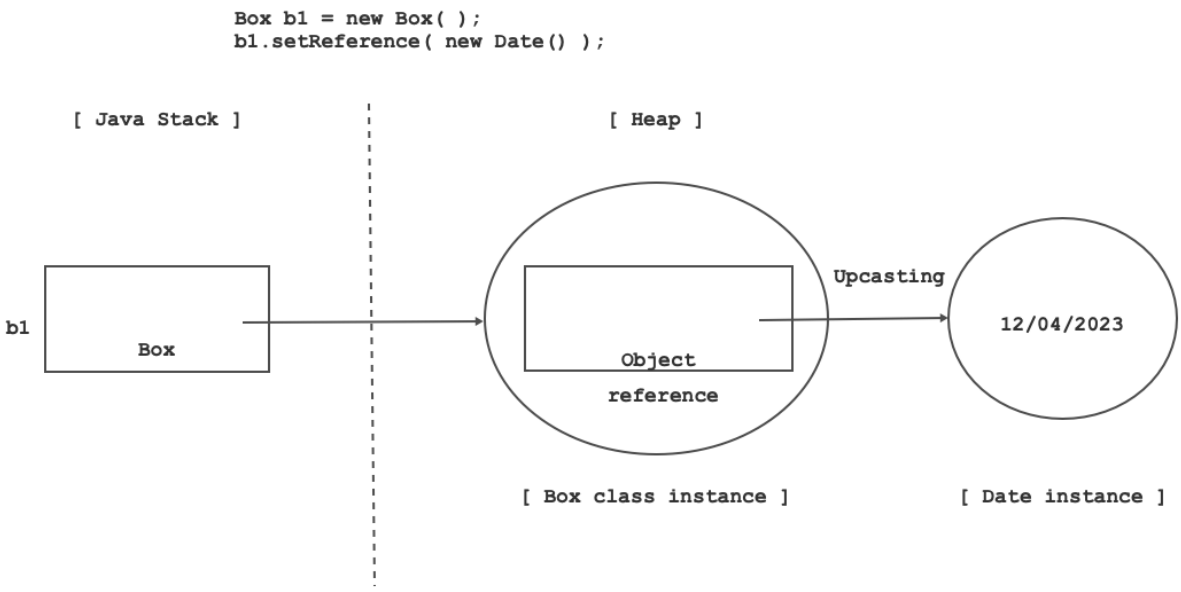
- Consider following code:

```
class Box{
  private Object reference;
  public Object getReference() {
    return reference;
  }
  public void setReference(Object reference) {
    this.reference = reference;
```

```
    }
}
```


Box b1 = new Box( );
[Java Stack]
[Heap]
b1
Box
null
Object
reference
[Box class instance]


Box b1 = new Box( );
b1.setReference(123);
[Java Stack]
[Heap]
b1
Box
Object
reference
Upcasting
123
[Box class instance]
[Integer instance]



- Using java.lang.Object class, we can write generic code but we can not write type-safe generic code.

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

- If we want to write type-safe generic code then we should use generics.

Generics

- Generics is a Java language feature which helps developer to write generic code by passing data type as a agrument.
- Consider generic code using Generics:

```
//Now class Box is called as parameterized type
class Box<T>{   // Here T is Type parameter
  private T reference;
  public T getReference() {
    return reference;
  }
  public void setReference(T reference) {
    this.reference = reference;
  }
}
public class Program {
  public static void main(String[] args) {

    Box<Date> b1 = new Box<Date>( );    //Here Date is called as Type
```

```
argument

    b1.setReference(new Date());

    Date date = b1.getReference();

    System.out.println(date);
  }
}
```

Why Generics

- Generics gives us stronger typechecking at compile time. In other words, using generics we can write type-safe generic code.
- It completly eliminates need of explict typecasting.
- It helps developer to define generic algorithms and data structures.

Generics Syntax:

- We can specify type argument during declaration of reference as well as instantiation.

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

- If we specify type argument during reference declaration then specifying type argument during instantiation is optional. It is called as type inference.

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

- We must specify type argument during declaration of reference.

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

- We can not use inheritance for type argument.

```
Box<Object> b1 = new Box< Date>( );   //Not OK
List<Date> list = new ArrayList<Date>( ); //OK
```

```
Box<Date> b1 = new Box< Object>( );   //Not OK
```

- If we use parameterized type without type argument then it is called as raw type.

```
Box b = new Box();    //OK: Here Box is called as raw type
//Box<Object> b1 = new Box< Object>( );
```

- During instantiation of parameterized type, type argument must be non primitive type.

```
Box<int> b1 = new Box<>( );   //Not OK: type argument int is not
allowed
```

- If we want to store primitive values inside instance of parameterized type then type argument must be Wrapper class.

```
Box<Integer> b1 = new Box<>( );   //OK
```

Commonly used type parameter names in Java

- T : Type
- N : Number
- K : Key
- V : Value
- E : Element
- S, U, R : Second type parameter names

We can specify multiple type arguments

```
interface Map<K, V>{
    K getKey( );
    V getvalue( );
}
class HashMap<K,V> implements Map<K,V>{
    private K key;
    private V value;
    public HashMap( 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) {
        Map<Integer, String> map = new HashMap<>(1, "DAC");
        Integer key = map.getKey();
        String value = map.getvalue();
        System.out.println(key+"    "+value);
    }
}
```

Bounded Type Parameter

- If we want to put restriction on data type which is used as type argument then we should specify bounded type parameter.

```
class Box<T extends Number>{    //Now T is bounded Type parameter
  private T reference;
  public T getReference() {
    return reference;
  }
  public void setReference(T reference) {
    this.reference = reference;
  }
}
public class Program {
  public static void main(String[] args) {
    //Box<Boolean> b1 = new  Box<>( );  //Not OK
    //Box<Character> b2 = new  Box<>( );    //Not OK
    Box<Number> b3 = new  Box<>( );     //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
  }
}
```

ArrayList demo

```
import java.util.ArrayList;

public class Program {
    public static void main(String[] args) {
        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));

        for( Integer element : list )
            System.out.println(element);
```

```
    }
}
```

Wild Card

- In Generics ? is called as wild card which represents unknown type.
- Types of wild card
    - Unbouned wild card
    - Upper Bouned wild card
    - Lower Bounded wild card.
- Consider following code

```
public static ArrayList<Integer> getIntegerArrayList( ) {
  ArrayList<Integer> list = new ArrayList<>();
  list.add(10);
  list.add(20);
  list.add(30);
  return list;
}
public static ArrayList<Double> getDoubleArrayList( ) {
  ArrayList<Double> list = new ArrayList<>();
  list.add(10.1);
  list.add(20.2);
  list.add(30.3);
  return list;
}
public static ArrayList<String> getStringArrayList( ) {
  ArrayList<String> list = new ArrayList<>();
  list.add("DAC");
  list.add("DMC");
  list.add("DESD");
  return list;
}
```

Unbouned 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.getIntegerArrayList();
      Program.printRecord( integerList );

      ArrayList<Double> doubleList = Program.getDoubleArrayList();
      Program.printRecord(doubleList);

      ArrayList<String> stringList = Program.getStringArrayList();
      Program.printRecord(stringList );
    }
```

Upper Bouned wild card

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

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

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

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

     ArrayList<String> stringList = Program.getStringArrayList();
     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.getIntegerArrayList();
    Program.printRecord( integerList ); //OK

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

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

Restrictions on Generics

- Reference: https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html