PIC 20A
Number, Autoboxing, and Unboxing

David Hyde
UCLA Mathematics

Last edited: May 15, 2020

Illustrative example

Consider the function that can take in any object.

```java
public static void printClassAndObj(Object obj) {
  System.out.println(obj.getClass());
  System.out.println(obj);
}
```

This method unfortunately doesn't work with primitives. Or does it?

(We'll talk about `getClass()` once we learn about generics.)

Illustrative example

```java
public static void main(String... args) {
  Complex c1 = new Complex(3.3,2.9);
  int i1 = 2;
  printClassAndObj(c1);
  printClassAndObj(i1);
}
```

The output is

```
class Complex
3.3+2.9i
class java.lang.Integer
2
```

Why did this work?

Illustrative example

In this line,

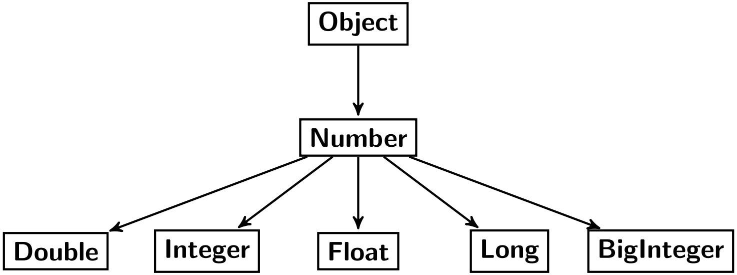
```
printClassAndObj(i1);
```

the `int` `i1` was *autoboxed* into an object of type `Integer`.

The Java language provides their respective wrapper `classes` for the 8 primitive types:
`boolean`, `byte`, `char`, `float`, `int`, `long`, `short`, and `double` correspond to
`Boolean`, `Byte`, `Character`, `Float`, `Integer`, `Long`, `Short`, and `Double`.

class Number

The `class java.lang.Number` and its subclasses has the following inheritance hierarchy.



(This figure is not complete. There are other subclasses of `Number` like `Short` or `BigDecimal`.)

class Number

The class Number is an abstract class designed to be inherited by classes representing numbers.

Number has the following abstract methods

```
public abstract double doubleValue()
```

```
public abstract float floatValue()
```

```
public abstract int intValue()
```

```
public abstract long longValue()
```

Any concrete subclass must provide an implementation of thses methods.

https://docs.oracle.com/javase/8/docs/api/java/lang/Number.html

class Number

`Number` provides the following concrete methods

```java
public byte byteValue()
```

```java
public short shortValue()
```

Subclasses of `Number` don't have to override these methods.

class Number

The implementations of `byteValue` and `shortValue` reveal why they can be provided as concrete methods.

```java
public byte byteValue() {
  return (byte) intValue();
}
```

```java
public short shortValue() {
  return (short) intValue();
}
```

`http://www.docjar.com/html/api/java/lang/Number.java.html`

class Number

Out of the 8 wrapper classes for primitive types, 6 inherit `Number`: `Byte`, `Float`, `Integer`, `Long`, `Short`, and `Double`. (`Boolean` and `Charater` do not inherit `Number`.)

These provide many useful features in addition to being an object that contains the primitive data type.

class Character

The class Character only inherits from Object. In addition to being the wrapper class of char, Character provides other useful features.

```
System.out.println(Character.toUpperCase('c'));
```

(toUpperCase is a static method.)

class Boolean

The `class` `java.lang.Boolean` only inherits from `Object`. It's not very useful aside from being the wrapper `class` of `boolean`.

Autoboxing and unboxing

The Java language provides special support to the 8 wrapper classes in the form of autoboxing and unboxing.

Autoboxing allows you to convert a primitive type into its respective `wrapper` class type, implicitly or explicitly.

```
boolean b1 = false;
Boolean b2 = b1;
Boolean b3 = (Boolean) b1;
```

Autoboxing and unboxing

Unboxing allows you to convert a wrapper `class` type into its respective primitive type, implicitly or explicitly.

```
Integer i1 = new Integer(3);
int i2 = i1;
int i3 = (int) i1;
```

Autoboxing and unboxing are special features provided by the language; you cannot make `classes` you write support autoboxing and unboxing.

Conclusion

Autoboxing and unboxing is slower than using primitive types but they can provide certain convenience.

```
import java.util.*;
public class Test {
  public static void main(String[] args) {
    ArrayList<Character> l = new ArrayList<>();
    //.add works because of autoboxing
    l.add('a'); l.add('b'); l.add('c');
    Collections.shuffle(l);

    for (int i=0; i<3; i++)
      System.out.println(l.get(i));
  }
}
```

You can use `Collections` with `Objects` but not with primitive types.