Wrappers
Lecture 17

Prof. Anita Agrawal

BITS- Pilani,K.K.Birla Goa campus

Wrappers, what and why???

- Java uses primitive types (also called as simple types) such as int, double etc..to hold the basic data types supported by the language.
- Primitives are not objects.
- Primitives yield better performance, by cutting down the overhead required if making objects of these even for the simplest of calculations.

Wrapper class, what and why??

- While storing in data structures which support only objects (ex. Array list and vectors), it is required to convert the primitive type to object first.

- Objects are needed if we wish to use the reference way of calling a method (because primitive types are passed by value).

- The classes in java.util package handle only objects
- ………
- Java handles this by using a wrapper class.
- A Wrapper class is a class which encapsulates a primitive type within an object.

- When we create an object of a wrapper class, it contains a field and in this field, we can store a primitive data type. In other words, we can wrap a primitive value into a wrapper class object.

- The eight classes of *java.lang* package are known as wrapper classes in java.

The eight classes of *java.lang* package are known as wrapper classes in java.

Primitive Type	**Wrapper class**
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Mechanisms

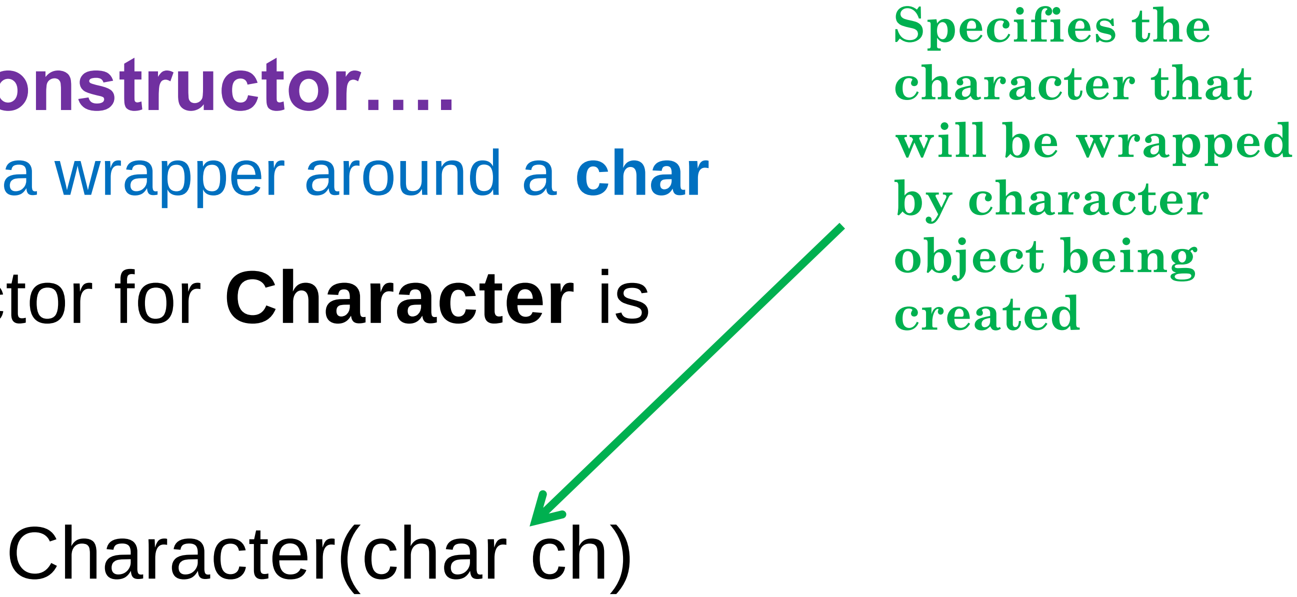
- Two mechanisms are possible:
- Boxing
- Unboxing
- Boxing: conversion of primitive into an object/ The process of encapsulating a primitive value within an object
- Unboxing: conversion of object into primitive/The process of extracting a primitive value from a type wrapper

Wrapper class

- Every wrapper class has two constructors,
  - First constructor takes corresponding primitive data as an argument
  - Second constructor takes string as an argument.
- The string passed to second constructor should be parse-able to number, otherwise you will get run time NumberFormatException.
- Wrapper Class Character has only one constructor which takes char type as an argument.
  - It doesn’t have a constructor which takes String as an argument
  - Because, String can not be converted into Character.
- Wrapper class Float has three constructors
  - The third constructor takes double type as an argument.

Constructors

- **Character constructor….**
  - **Character** is a wrapper around a **char**
- The constructor for **Character** is



Character(char ch)

Specifies the character that will be wrapped by character object being created

Boolean constructors

- **Boolean** is a wrapper around **boolean** values
- It defines the following constructors:
  - Boolean(boolean boolValue) // boolValue must be either true or false
  - Boolean(String boolString) // If *boolString* contains the string “true” (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

Integer constructors…

- int value
- Integer(int num)

- String value
  - Integer(String str)
    - If *str* does not contain a valid numeric value, then a N**umber Format Exception** is thrown.

Boxing and Unboxing: Method 1

- Example: Wrappers.java

Numeric type Wrappers

- The most commonly used wrappers are the numeric type.
- All of the numeric type wrappers inherit the abstract class **Number**
- **Number** declares methods that returns the primitive value of an object in each of the different number formats
- These methods are implemented by each of the numeric type wrappers

byte byteValue( )

double doubleValue( )

float floatValue( )

int intValue( )

long longValue( )

short shortValue( )

Boxing and Unboxing (Method 2)

- To obtain the int value contained in an Integer object

```
int intValue( ) ;
```

//Returns the encapsulated int value

- To obtain the **char** value contained in a **Character** object

```
char charValue( ) ;
```

//Returns the encapsulated character

- To obtain a boolean value from a **Boolean** object
  - boolean booleanValue( )

etc…

Boxing and Unboxing

- All of the type wrappers override **toString( )**
- Return the human-readable form of the value contained within the wrapper

  Output the value by directly passing a type wrapper object to println( ) without converting into its primitive type
- Integer x= new Integer(15); //boxing
- int i = x.intValue();   //unboxing
- System.out.println(i+ “ “ + x); // displays 15   15

Autobox and Autounbox(Method 3)

- From J2SE 5.0, two additional features were introduced:

(i) Autoboxing

(ii) AutoUnboxing

(i) Autoboxing: A primitive type is **automatically** encapsulated (boxed) into its equivalent type wrapper (whenever an object is needed)

- No longer necessary to manually construct an object

(ii) Auto-Unboxing: The value of a boxed object is **automatically** extracted (unboxed) from a type wrapper (when its value is needed)

- No need to call a method such as **intValue( )** or **doubleValue()** etc..

Autoboxing

- You need only assign that value to a type-wrapper reference
- Java automatically constructs the object for you
- Example:

```
    class AutoBox {
       public static void main(String args[ ]) {
         Integer j= 15;                    //autoboxing
         int i= j;                         //autoUnboxing
      System.out.println(i+ " " + j);      //Displays 15  15
}
}
```

Autoboxing and methods

- Autoboxing/Auto-unboxing can also occur when

  - An argument is passed to a method

  - When a value is returned by a method

Example:

```
class AutoBox2 {
        static int mat(Integer v) {
                return v ;                                  // autounbox
                        }
      public static void main(String args[]) {
                Integer j1= mat (15);                       // autobox
                System.out.println(j1);                      // 15
                                }
                        }
```

AutoBoxing/AutoUnboxing in Expressions

- In general, auto-boxing/unboxing take place whenever a conversion into an object or from an object is required.
- Example:
- Integer i1, i2;

```
  i1= 90;              //Autoboxing
  ++i1;                //Auto-unboxing and then reboxing
i2 = i1 + (i1/ 3);     //Auto-unboxing and then reboxing
int i = i1+ (i1/ 3);   //Auto-unboxing and NO reboxing
```

AutoBoxing/AutoUnboxing in Expressions

- Auto-unboxing also allows you to mix different types of numeric objects in an expression
- Once the values are unboxed, the standard **type promotions** and conversions are applied
- You can even use **Integer** numeric objects to control a **switch** statement

AutoBoxing/AutoUnboxing in Expressions

```
Integer i1= 18;
Double d1= 50.5;
d1= d1+ i1;

Integer i2= 2;
switch(i2) {
case 1: System.out.println("one");
break;
case 2: System.out.println("two");
break;
default: System.out.println("error");}
```

Autoboxing/AutoUnboxing Boolean and character values

```
Boolean b = true;                    //Autobox a boolean
if(b)
{
System.out.println("b is true");
 }
Character ch= 'x';                   // box a char
char ch2 = ch;                             // unbox a char
```

When to use objects…

- Should we use objects such as **Integer** or **Double** exclusively, abandoning primitives altogether?
- Example:

```
Double a, b, c;
a = 10.0;
b = 4.0;
c = Math.sqrt(a*a + b*b);
System.out.println("Hypotenuse is " + c);
```

- Far less efficient than the equivalent code written using the primitive type **double**
- Restrict the use of the type wrappers **to only** those cases in which an object representation of a primitive type is required