Java Basics, Strings

STRINGS - INTRO, CREATION

What is a String?

String → a sequence of characters

Example of **strings:**

- "The cow jumps over the moon"
- "PRESIDENT OBAMA"
- "12345"

What is a **String class** in Java?

- String class in Java: holds a "sequence of characters"

String **greeting** = new String("Hello world!");

- Creating new String object
- Contains sequence of chars

String Class

- Why use String class vs. char array?
- **Advantage:** provides many useful methods for string manipulation
  - Print length of string – **str.length()**
  - Convert to lowercase – **str.toLowerCase()**
  - Convert to uppercase – **str.toUpperCase()**

  * Many others

Creating String Objects

There are 2 ways to create String objects

Method 1: String **greeting1** = **new** String("Hello World!") ;

- **Variable** of **type String**
- **Name** of variable is **greeting1**

- **new** operator for creating instance of the **String** class
- **Recall:** Instance of a class is an object

- **String value** aka *string literal*
- **String literal:** series of characters enclosed in double quotes.

Creating String Objects

There are 2 ways to create String objects

Method 2: String **greeting2** = “Hello World Again!” ;

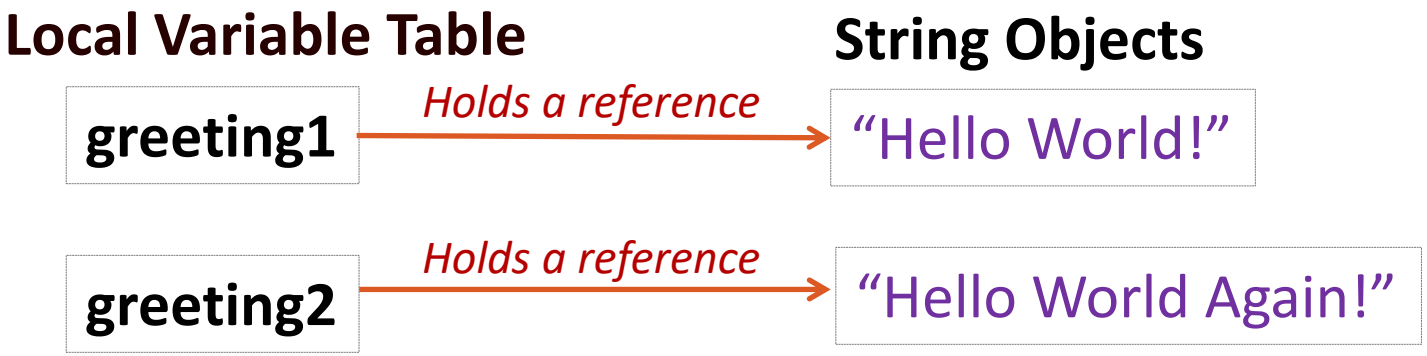
- **Shorthand** for String creation (most used)
- **Behind the scenes:** new instance of String class with “Hello World Again!” as the value

Creating String Objects

There are 2 ways to create String objects

Method 1: String **greeting1** = **new** String("Hello World!") ;

Method 2: String **greeting2** = "Hello World Again!" ;



String Constructor

- **Recall:** when new object created ➔ the constructor method is always called first
- Pass initial arguments or empty object
- String class has multiple constructors

String Constructor

```
String str1= new String() ; //empty object
```

```
String str2= new String("string") ; //string input
```

```
String str3= new String(char[]) ; //char array input
```

```
String str4= new String(byte[]) ; //byte array input
```

* few others

Strings: Defining and initializing

Simple example

```
String s1 = "Welcome to Java!";
String s2 = new String("Welcome to Java!"); //same as s1
```

Numbers as strings

```
String s3 =“12345”;
String s4 = new String(s3); //s4 will hold same value as s3
```

Char array as strings

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String s5= new String(helloArray);
```

Strings: Defining and initializing

Empty Strings

String **s5** = "";

String **s6** = new String(“”);

String object created;
String value is empty “”

Null String

String **s7** = null;

String variable not pointing to any String object

Understanding String Creation

```
String greeting1 = "Hello !"
String greeting2 = "Hello !"
```

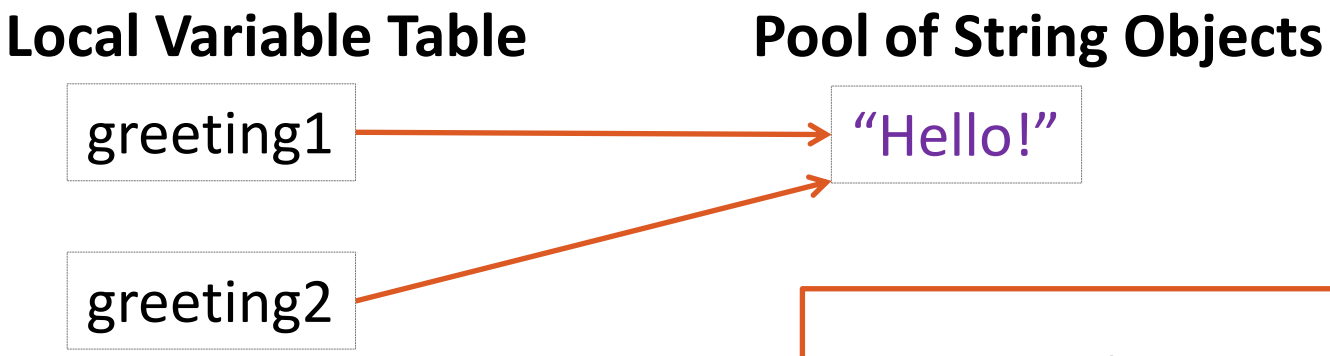
Does Java create 2 String objects internally?

Without **"new"** operator for String creation:

- Java looks into a String pool (collection of String objects)
  - Try to find objects with same **string** value
- If **object exists** ➔ new variable points to existing object
- If **object does not exist** ➔ new object is created
- Efficiency reasons – to limit object creation

Understanding String Creation

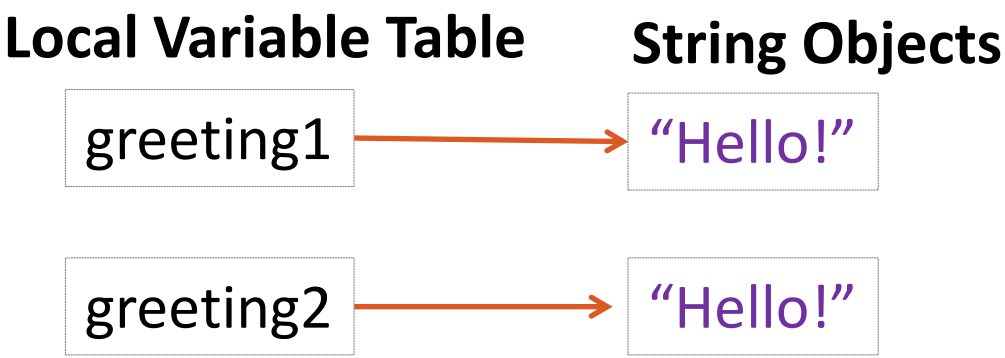
String **greeting1** = “Hello!”
String **greeting2** = “Hello!”



Concept of String pooling

Understanding String Creation

```
String greeting1 = new String ("Hello!");
String greeting2 = new String ("Hello!");
```



STRINGS - METHODS, CONCATENATION

String Methods

Advantage of String class: many built-in methods for String manipulation

```
str.length();          // get length of string
str.toLowerCase()      // convert to lower case
str.toUpperCase()      // convert to upper case
str.charAt(i)          // what is at character i?
str.contains(..)       // String contains another string?
str.startsWith(..)     // String starts with some prefix?
str.indexOf(..)        // what is the position of a character?
....many more
```

String Methods - length, charAt

str.length() ➔ Returns the **number of chars** in String

str.charAt(i) ➔ Returns the **character at position** i

Character positions in strings are numbered starting from 0 – just like arrays

String Methods - length, charAt

str.length() ➔ Returns the **number of chars** in String

str.charAt(i) ➔ Returns the **character at position** i

Returns:

```
"Utah".length();
```

4

```
"Utah".charAt (2);
```

a

0123

String Methods – valueOf(X)

String.valueOf(**X**) - Returns **String representation** of **X**

- ***X: char, int, char array, double, float, Object***
- Useful for converting different data types into String

```
String str1 = String.valueOf(4);    //returns "4"
```

```
String str2 = String.valueOf('A');  //returns "A"
```

```
String str3 = String.valueOf(40.02); //returns "40.02"
```

String Methods – substring(..)

str.substring(..) ➔ returns a new String by copying characters from an existing String.

- str.**substring** (**i**, **k**)
  - returns substring of chars from pos i to k-1
- str.**substring** (**i**);
  - returns substring from the i-th char to the end

String Methods – substring(..)

Returns:

"Ben".substring(**0**,**2**); → **"Be"**

012

"John".substring(**1**); → **"ohn"**

0123

"Tom".substring(**9**); → **""** **(empty)**

012

String Concatenation – Combine Strings

- What if we wanted to combine String values?

  String **word1** = “re”;
  String **word2** = “think”;
  String **word3** = “ing”;

  How to combine and make ➔ **“rethinking” ?**

- Different ways to concatenate Strings in Java

String Concatenation – Combine Strings

String **word1** = “re”;
String **word2** = “think”;
String **word3** = “ing”;

Method 1: Plus “+” operator

String **str** = word1 **+** word2;

– *concatenates word1 and word2* ➔ ***“rethink”***

Method 2: Use String’s “concat” method

String **str** = word1.**concat** (word2);

– *the same as word1 + word2* ➔ ***“rethink”***

String Concatenation – Combine Strings

Now **str** has value **"rethink",** how to make **"rethinking"**?

String **word3** = "ing";

Method 1: Plus "+" operator

```
str = str + word3;  //results in "rethinking"
```

Method 2: Use String's "concat" method

```
str = str.concat(word3);  //results in "rethinking"
```

Method 3: Shorthand

```
str += word3;  //results in "rethinking" (same as method 1)
```

String Concatenation: Strings, Numbers & Characters

String **myWord**= “Rethinking”;
int **myInt=**2;
char **myChar**=‘!’;

Internally: myInt & myChar converted to String objects

Method 1: Plus “+” operator

```
String result = myWord + myInt + myChar;
//Results in “Rethinking2!”
```

String Concatenation: Strings, Numbers & Characters

```
String myWord= "Rethinking";
int  myInt=2;
char myChar='!';
```

Method 2: Use String's "concat" method

```
String strMyInt= String.valueOf(myInt);
String strMyChar=String.valueOf(myChar);
String result = myWord.concat(strMyInt).concat(strMyChar);
//Results in "Rethinking2!"
```

STRING IMMUTABILITY

String Immutability

- Strings in Java are immutable
- **Meaning:** cannot change its value, once created

```
String str;

str = "Java is Fun!";
str = "I hate Java!";
```

Did we change the value of **"Java is Fun!"** to **"I hate Java!"**?

String Immutability

```
String str;

str = "Java is Fun!";
str = "I hate Java!";
```

Cannot insert or remove a value

e.g. remove "Fun"

Local Variable Table

str

String Objects

"Java is Fun!"

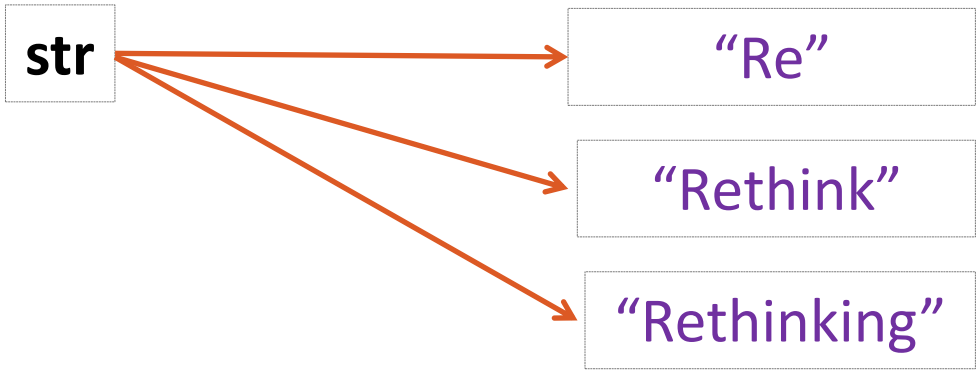
"I hate Java!"

String Immutability

```
String str;
str  = "Re";
str =  str + "think"; //Rethink
str =  str + "ing";   // Rethinking
```

Every concat: Create new String object
Unused objects: "Re", "Rethink" go to garb. coll.

Local Variable Table | **String Objects**



String Immutability

- **Problem:** With frequent modifications of Strings
  - Create many new objects – uses up memory
  - Destroy many unused ones – increase JVM workload
  - **Overall:** can slow down performance

- **Solution** for frequently changing Strings:
  - StringBuilder Class instead of String

StringBuilder Class

- StringBuilders used for String concatenation

- StringBuilder class is "**mutable**"
  Meaning: values within StringBuilder can be changed or modified as needed

- In contrast to Strings where Strings are immutable

StringBuilder - “append” method

- append(X) ➔ most used method
  - Appends a value **X** to end of StringBuilder
  - ***X: int, char, String, double, object (almost anything)***

StringBuilder for String Construction

```
StringBuilder sb = new StringBuilder(); //obj creation
sb.append("Re");      //add "Re" to sb
sb.append("think"); //add "think" to sb
sb.append("ing");     //add "ing" to sb
String str= sb.toString(); //return String va
```

- **Only 1 object**
- **All 3 words appended to same object**

Local Variable Table **StringBuil Object**

Bottom-line: *Use StringBuilder when you have frequent string modification*

STRING EQUALITY/INEQUALITY

Testing String Equality

- How to check if two Strings **contain same value**?

```
String str1=new String("Hello World!");
String str2=new String("Hello World!");

if(str1==str2) { //eval to false
  System.out.println("same");
}
```

if **str1** **referencing** same object as **str2**?

Local Variable Table **String Objects**

str1 → "Hello World!"

str2 → "Hello World!"

Testing String Equality

- How to check if two Strings **contain same value**?

```
String str1=new String("Hello World!");
String str2=new String("Hello World!");

if(str1==str2) { //eval to false
  System.out.println("same")
}

if(str1.equals(str2)) { //eval to true
  System.out.println("same");
}
```

if **content** of str1 same as str2?

Testing String Equality

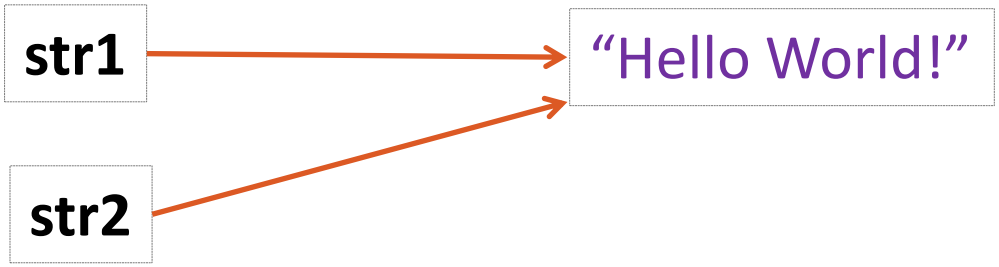
- What if "new" operator not used?

```
String str1 = "Hello World!";
String str2 = "Hello World!";

if(str1==str2) { //eval to true
  System.out.println("same");
}
```

if **str1 referencing** same object as **str2**?

Local Variable Table **String Objects**



Testing String Equality

- What if “new” operator not used?

```
String str1 = “Hello World!”;
String str2 = “Hello World!”;

if(str1==str2) { //eval to true
  System.out.println(“same”);
}

if(str1.equals(str2)) { //eval to true
  System.out.println(“same”);
}
```

Testing String Equality

- **Point to note:** String variables are references to String objects (i.e. memory addresses)
- **"str1==str2"** on String objects compares memory addresses, not the contents
- Always use **"str1.equals(str2)"** to compare contents

Java StringBuffer class

- Java StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.
- Important Constructors of StringBuffer class
- **StringBuffer():** creates an empty string buffer with the initial capacity of 16.
- **StringBuffer(String str):** creates a string buffer with the specified string.
- **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.

length( ) and capacity( )

- The current length of a **StringBuffer can be found via the length( ) method, while the total** allocated capacity can be found through the **capacity( ) method. They have the following**
- general forms:
- int length( )
- int capacity( )

- Here is an example:
- // StringBuffer length vs. capacity.
- class StringBufferDemo {
- public static void main(String args[]) {
- StringBuffer sb = new StringBuffer("Hello");
- System.out.println("buffer = " + sb);
- System.out.println("length = " + sb.length());A
- System.out.println("capacity = " + sb.capacity());
- }
- }

```
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
sb.insert(1,"J");//now original string is changed
System.out.println(sb);//prints
sb.replace(1,3,"Jav");
System.out.println(sb);
sb.delete(1,3);
System.out.println(sb)
```

WRAPPER CLASSES (SIDE TOPIC)

Wrapper Class in Java

- Java is not a purely object-oriented programming language, the reason being it works on primitive data types.
- These eight primitive data types int, short, byte, long, float, double, char and, boolean are not objects.
- We use wrapper classes to use these data types in the form of objects.
- Wrapper class in Java makes the Java code fully object-oriented.
- For example, converting an int to Integer. Here int is a data type and Integer is the wrapper class of int.

- Eight wrapper classes exist in **java.lang** package that represent 8 data types. Following list gives.

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

Need for Wrapper class in Java Adv

- Wrapper classes are used to provide a mechanism to ‘wrap’ or bind the values of primitive data types into an object. This helps primitives types act like objects and do the activities reserved for objects like we can add these converted types to the collections like ArrayList, HashSet, HashMap, etc.
- Wrapper classes are also used to provide a variety of utility functions for primitives data types like converting primitive types to string objects and vice-versa, converting to various bases like binary, octal or hexadecimal, or comparing various objects.
- We can not provide null values to Primitive types but wrapper classes can be null. So wrapper classes can be used in such cases we want to assign a null value to primitive data types.

Converting primitive numbers to Object numbers using constructor methods

Constructor calling	Conversion Action
Integer IntVal = new Integer(i);	Primitive integer to Integer object
Float FloatVal = new Float(f);	Primitive float to Float object
Double DoubleVal = new Double(d);	Primitive double to Double object
Long LongVal = new Long(l);	Primitive long to Long object

Creating wrapper objects

- The most straightforward way to create a wrapper object is to use its constructor:
- Integer i = new Integer (17); or
- Integer i = 17;
- Double d = new Double (3.14159);
- or Double d = 3.14159;
- Character c = new Character ('b');
- or Character c = ‘b’;

Extracting the values

- Java knows how to print wrapper objects, so the easiest way to extract a value is just to print the object or assign it to a primitive data type.
- Integer i = new Integer (17);
- Double d = new Double (3.14159); System.out.println (i);
- System.out.println (d);

- You can use the toString method to convert the contents of the wrapper object to a String

- String iStr = i.toString();
- String dStr = d.toString();

- To extract the primitive value from the object, there is an object method in each wrapper class that does the job:
- int iValue = i.intValue(); or
- int iValue = i;
- double dValue = d.doubleValue();
- or double dValue = d;

- There are also methods for converting wrapper classes into different primitive types.

Java Autoboxing - Primitive Type to Wrapper Object

- **In autoboxing, the Java compiler automatically converts primitive types into their corresponding wrapper class objects. For example,**

- **int a = 56;**

- **// autoboxing**
- **Integer aObj = a;**

- **Autoboxing has a great advantage while working with Java collections.**

- Creating Wrapper Objects
- To create a wrapper object, use the wrapper class instead of the primitive type. To get the value, you can just print the object:

Example

```
public class Main {
  public static void main(String[] args) {
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
    System.out.println(myInt);
    System.out.println(myDouble);
    System.out.println(myChar);
  }
}
```

Java Unboxing - Wrapper Objects to Primitive Types

- In unboxing, the Java compiler automatically converts wrapper class objects into their corresponding primitive types. For example,

```
// autoboxing
Integer aObj = 56;
// unboxing
int a = aObj;
```

Converting Numeric Strings to Primitive numbers using Parsing method

Method calling	Conversion Action
int i = Integer.parseInt(str);	Converts String str into primitive integer i
long l = Long.parseLong(str);	Converts String str into primitive long l

Byte

The Byte class encapsulates a byte value. It defines the constants MAX_VALUE and MIN_VALUE and provides these constructors:

Byte(**byte b**)
Byte(**String str**)

Here, b is a byte value and str is the string equivalent of a byte value.

Creating a Byte object

```
class A
{
public static void main(String... ar)
{
Byte b1 = new Byte((byte)10); //casting the default int value to byte.
Byte b2 = new Byte("10"); //Passing primitive byte as a String.
}
}
```

Methods	Description
int **compareTo**(Byte b)	- Returns a zero if the invoked Byte object contains the same byte value as b. - Returns a positive value if the invoked Byte object contains greater value than b. - Returns a negative value if the invoked Byte object contains smaller value than b.
boolean **equals**(Object ob)	Returns a true if the invoked Byte object has same value as referred by ob, else false.
static *byte* **parseByte**(String s)	Returns a primitive byte value if String s could be converted to a valid byte value.
static *Byte* ***valueOf****(byte b)*	Returns a Byte object after converting it from primitive byte value, b.
short **shortValue()**	Returns an primitive short value after converting it from an invoked Byte object.
byte **byteValue()**	Returns an byte value after converting it from an invoked Byte object.
int **intValue()**	Returns an int value after converting it from an invoked Byte object.
long **longValue()**	Returns an long value after converting it from an invoked Byte object.

//Converting Byte to short, int, long, float, double

- class A
- {
- public static void main(String... ar)
- {
- Byte y = new Byte("10"); // Converting a String argument to wrapped Integer object

- System.out.println("Value in wrapped object,y "+ y);

- byte b = y.byteValue(); //Returns a primitive byte value out of a wrapped Byte object
- short s= y.shortValue(); //Returns a primitive short value out of a wrapped Byte object
- int i = y.intValue(); //Returns a primitive int value out of a wrapped Byte object
- long l = y.longValue(); //Returns a primitive long value out of a wrapped Byte object
- float f = y.floatValue(); //Returns a primitive float value out of a wrapped Byte object

- ***To convert a String to a primitive byte value using parseByte() method.***
- Method **parseByte()**, converts a string value which could be parse to a primitive byte value.

- **byte b1 = Byte.parseByte("10");**
- **byte b2 = Byte.parseByte("100");**
- **byte b3 = Byte.parseByte("50");**
- **System.out.println("Primitive byte value in b1 : "+ b1);**
  System.out.println("Primitive byte value in b2 : "+ b2);
  System.out.println("Primitive byte value in b3 : "+ b3); }

Character Wrapper Class

Constructor	Description
Character(char c)	Constructor of Character wrapper class only takes a primitive **char** value.

Methods	Description
static *boolean* **isDigit**(char ch)	Returns a true if ch is digit else, false.
static *boolean* **isLetter**(char ch)	Returns a true if ch is a letter, else false.
static *boolean* **IsLetterOrDigit**(char ch)	Returns a true if ch is either a letter or a digit, else false.
static *boolean* **isLowerCase**(char ch)	Returns true if ch is a lowercase letter, else false
static *boolean* **isUpperCase**(char ch)	Returns true if ch is an uppercase letter, else false.
static *boolean* **iswhite space**(char ch)	Returns a true if a ch is a white space character, else false.
static *char* **toLowerCase**(char ch)	returns the lowercase form of ch.
static *char* **toUpperCase**(char ch)	Returns an uppercase form of ch.

```
import java.util.*;

class A
{
public static void main(String args[])
{
        System.out.println("Is char a digit? : " + Character.isDigit('a'));
        System.out.println("Is char a digit? : " + Character.isDigit('1'));
        System.out.println("Is char a letter? : " + Character.isLetter('a'));
        System.out.println("Is char a letter? : " + Character.isLetter('1'));
        System.out.println("Is char a letter or a digit? : " + Character.isLetterOrDigit('-'));
        System.out.println("Is char a letter or a digit? : " + Character.isLetterOrDigit('1'));
        System.out.println("Is char a white space? : " + Character.iswhite space(' '));
}
}
```

Output-

Is char a digit? : false
Is char a digit? : true
Is char a letter? : true
Is char a letter? : false
Is char a letter or a digit? : false
Is char a letter or a digit? : true
Is char a white space? : true

- ***Constructor of Integer wrapper class***

Constructor	Description
Integer(int i)	Constructor of Integer wrapper class takes a primitive **int** value.
Integer(String str)	Constructor of Integer wrapper class also takes a **String** equivalent of a primitive int value.

• Creating an Integer object

```
class A
{
public static void main(String... ar)
{
Integer b1 = new Integer(10);          //Passing a primitive int value.
Integer b2 = new Integer("20");        //Passing primitive int as a String.
}
}
```

Some important methods of Integer wrapper class

Methods	Description
int **compareTo**(Integer b)	Returns a zero if the invoked Integer object contains the same value as b. Returns a positive value if the invoked Integer object contains greater value than b. Returns a negative value if the invoked Integer object contains smaller value than b.
boolean **equals**(Object ob)	Returns a true if invoked Long object has same value as referred by ob, else false.
static *int* **parseInteger**(String s)	Returns a primitive int value if String, s could be converted to a valid int value.
static *Integer* ***valueOf****(int b)*	Returns an Integer object after converting it from a primitive int value, b.
static *Integer* **valueOf**(String s)	Returns a Integer object after converting it from a String, s.
short **shortValue**()	Converts a Integer object to a primitive short value and returns it.
byte **byteValue**()	Converts an Integer object to a primitive byte value and returns it.

int **intValue()**	Converts an Integer object to a primitive int value and returns it
long **longValue()**	Converts an Integer object to a primitive long value and returns it
float **floatValue()**	Converts an Integer object to a primitive float value and returns it
double **doubleValue()**	Converts an Integer object to a primitive double value and returns it.

- ***Using compareTo() method to compare values in two Integer objects.***
- Method **compareTo(Integer i)** takes **Integer class** type object and
- it -Returns a **zero** if the invoked Integer object contains the value **same as i**.
- Returns **1** if the invoked Integer object contains value **larger than i**.
- Returns **-1** if the invoked Integer object contains value **smaller than i**.

```
class A
{
public static void main(String... ar)
{
Integer i1 = new Integer("10"); //Constructor accepting String value
Integer i2 = new Integer(10);    //Constructor accepting primitive int value

System.out.println("Value in i1 = "+ i1);
System.out.println("Value in i2 = "+ i2);

System.out.println("Invoking i1 to compare with i2 : "+ i1.compareTo(i2));

Integer i3 = new Integer("11"); //Passing primitive int as a String to Constructor
Integer i4 = new Integer(20);  //Passing a primitive int directly to Constructor

System.out.println("Value in i3 = "+i3);
System.out.println("Value in i4 = "+i4);

System.out.println("Invoking i3 to compare with i4 : "+ i3.compareTo(i4));

System.out.println("Invoking i4 to compare with i3 : "+ i4.compareTo(i3));

}
}
```

```
//Converting String to primitive int value.

import java.util.*;

class A
{
public static void main(String... ar)
{
int b1 = Integer.parseInt("20");
int b2 = Integer.parseInt("-200");

System.out.println("Primitive int value in b1 : "+ b1);
System.out.println("Primitive int value in b2 : "+ b2);
}
}
```

Double Wrapper Class

Double wrapper class is used to create an object version of a *primitive double value*.

- ***Constructor of Double wrapper class***

Constructor	Description
Double(double d)	Constructor of Double wrapper class takes a primitive **double** value.
Double(String str)	Constructor of Double wrapper class also takes a **String** equivalent of a primitive double value..

byteValue()	It returns the value of Double as a byte after the conversion.
compare(double d1, double d2)	It compares the two double values.
compareTo(Double another Double)	It compares two Double objects numerically.
doubleValue()	It returns the value of Double as a double after the conversion.
equals(Object obj)	It compares the object with the specified object.
floatValue()	It returns the float type value for the given Double object.
hashCode()	It returns the hash code for the given Double object.
hashCode(Double value)	It returns the hash code for the given Double value.
intValue()	It returns the value of Double as an int after the conversion.

max(double a, double b)	It returns the greater of the two double values.
min(double a, double b)	It returns the smaller of the two double values.
parseDouble(String s)	It returns a new double which is initialized by the value provided by the String.
shortValue()	It returns the value of Double as a short after the conversion.
sum(double a, double b)	It adds the two values as per the + operator.
toHexString(double d)	It returns a hexadecimal string represented by the double argument.
toString()	It returns a string represented by the Double object.
toString(double d)	It returns a string represented by the

```
public class DoubleDemo1
{
    public static void main(String[] args)
    {
            double a = 46.23;
            System.out.println("toString(a) = " + Double.toString(a));
            double a = 46.10;
            Double obj = new Double(a);
            System.out.println("Double.toHexString(a) = " +
Double.toHexString(a));
    }
}
```

Wrapper Classes

- **Recall:** primitive data types int, double, long are not objects

- **Wrapper classes:** convert primitive types into objects
  - **int:** Integer wrapper class
  - **double:** Double wrapper class
  - **char:** Character wrapper class

- 8 Wrapper classes:
  - Boolean, Byte, Character, Double, Float, Integer, Long, Short

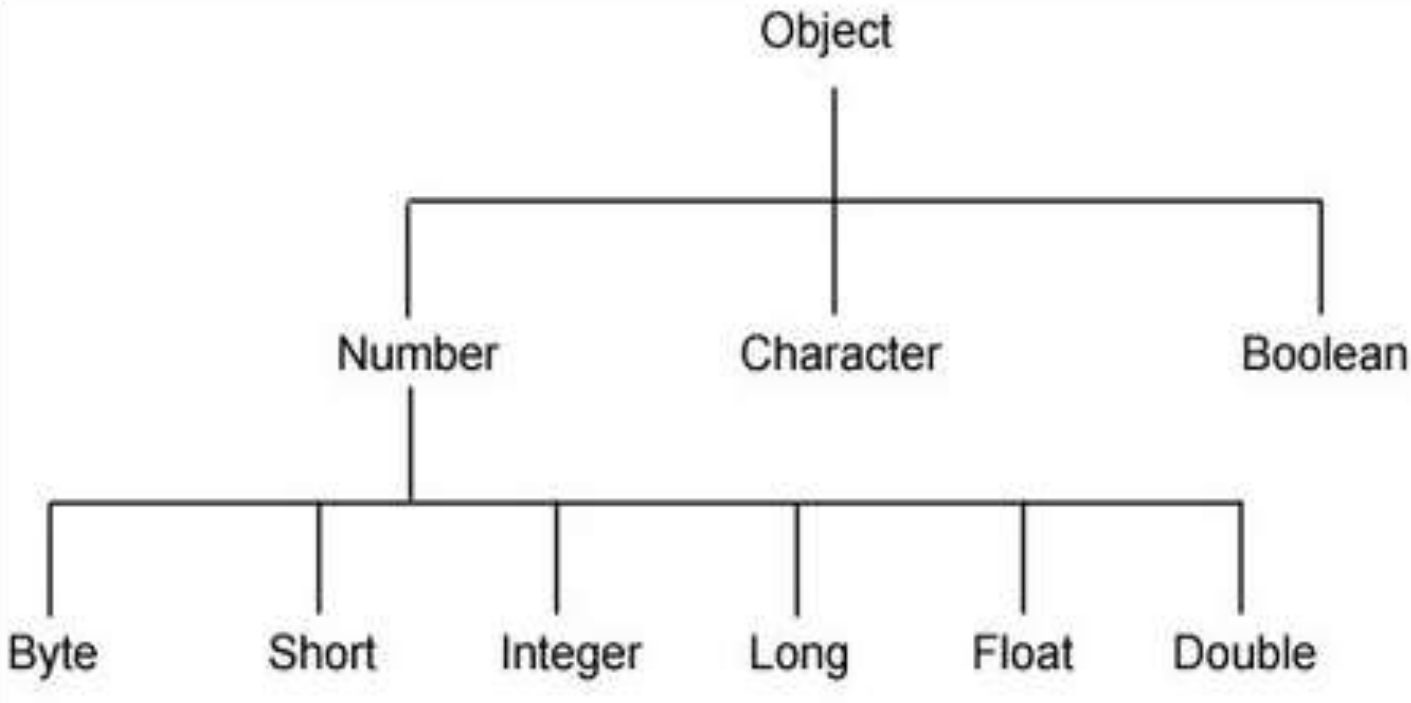


Figure : Wrapper classes Hierarchy

Wrapper Classes

- Why are they nice to have?
  - Primitives have a limited set of in-built operations
  - Wrapper classes provide many common functions to work with primitives efficiently
- How to convert a **String "22" ➔ int**?
  - Cannot do this with primitive type int
  - Can use Integer wrapper class; Integer.parseInt(..)

```
String myStr = "22";
int myInt= Integer.parseInt(myStr);
```

Wrapper Classes

- **Reverse:** How to convert **int 22 ➔ String**?
  - Cannot do this with primitive int
  - Can use Integer wrapper class; Integer.toString(..)

```
int myInt= 22;
String myStr= Integer.toString(myInt); // "22"
String myStr2 = String.valueOf(myInt); // "22"
```

equivalent

- Each Wrapper class has its own set of methods

Integer Wrapper Class Methods

- bitCount(int i) : int - Integer
- compare(int x, int y) : int - Integer
- decode(String nm) : Integer - Integer
- getInteger(String nm) : Integer - Integer
- getInteger(String nm, int val) : Integer - Integer
- getInteger(String nm, Integer val) : Integer - Integer
- highestOneBit(int i) : int - Integer
- lowestOneBit(int i) : int - Integer
- numberOfLeadingZeros(int i) : int - Integer
- numberOfTrailingZeros(int i) : int - Integer
- parseInt(String s) : int - Integer
- parseInt(String s, int radix) : int - Integer
- reverse(int i) : int - Integer
- reverseBytes(int i) : int - Integer
- rotateLeft(int i, int distance) : int - Integer
- rotateRight(int i, int distance) : int - Integer
- signum(int i) : int - Integer
- toBinaryString(int i) : String - Integer
- toHexString(int i) : String - Integer
- toOctalString(int i) : String - Integer
- toString(int i) : String - Integer
- toString(int i, int radix) : String - Integer

Character Wrapper Class Methods

```
charCount(int codePoint) : int - Character
codePointAt(char[] a, int index) : int - Character
codePointAt(CharSequence seq, int index) : int - Character
codePointAt(char[] a, int index, int limit) : int - Character
codePointBefore(char[] a, int index) : int - Character
codePointBefore(CharSequence seq, int index) : int - Character
codePointBefore(char[] a, int index, int start) : int - Character
codePointCount(char[] a, int offset, int count) : int - Character
codePointCount(CharSequence seq, int beginIndex, int endIndex) : int - C
compare(char x, char y) : int - Character
digit(char ch, int radix) : int - Character
digit(int codePoint, int radix) : int - Character
forDigit(int digit, int radix) : char - Character
getDirectionality(char ch) : byte - Character
getDirectionality(int codePoint) : byte - Character
getName(int codePoint) : String - Character
getNumericValue(char ch) : int - Character
getNumericValue(int codePoint) : int - Character
getType(char ch) : int - Character
getType(int codePoint) : int - Character
Press 'Ctrl+Space' to show Template Proposals
```

Double Wrapper Class Methods