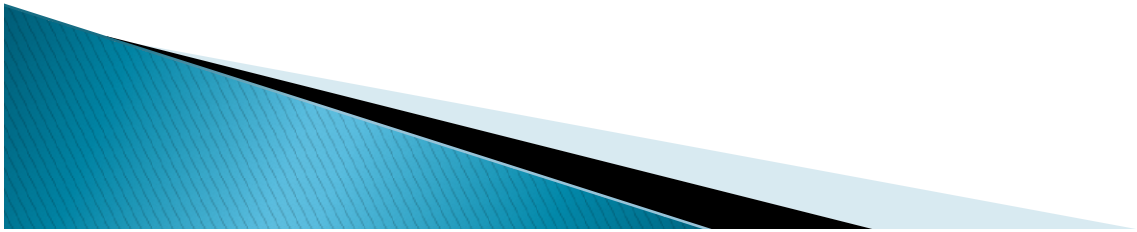
String Class in Java

By
Sunil Kumar(Master of Sc.)
Bangalore, India

1. Agenda

- String Class
- StringBuilder class
- Primitive wrapper classes
- Final fields
- Enumeration types



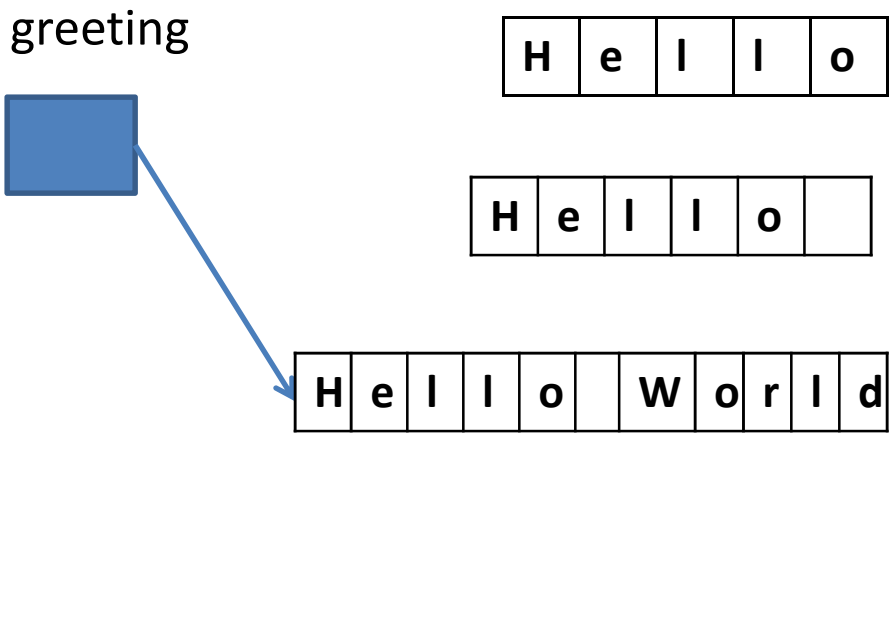
String Class

- In our earlier classes, we have talked about primitive data type and in that char type which allows us to work deal with individual characters. In general, we usually work with sequence of characters and so String class comes in.
- The String class stores a sequence of Unicode characters
  - Stored using UTF-16 encoding (Unicode Transformation Format).
  - Literals are enclosed in double quotes (“ “)
  - Values can be concatenated using + and +=
  - String object are immutable

String Class

- What does this immutable means.
  - Suppose we have declared the java program as below.

```
String greeting = “Hello”;
greeting += greeting + “ ”
greeting +=
greeting+”World”;
```



String Class

- After 1st line of executing the code, a reference is created with the name greeting and memory is allocated as shown in the figure.
- After 2nd line, another new memory area is created and now greeting is re-pointing to the newly created memory.
- Similarly after the 3rd line.

String Class Methods

Operation	Methods
Length	length
String for non-string	valueOf
Create new string(s) from existing	concat, replace, toLowerCase, toUpperCase, trim, split
Formatting	format
Extract substring	chatAt
Test substring	contains, endsWith, startWith, indexOf, lastIndexOf
comparison	compareTo, compareToIgnoreCase, isEmpty, equals, equalsIgnoreCase

String Class Methods

- These string class methods whatever are discussed in the last slides are just the common methods used frequently during the program.
- There are lot of other methods which can be found at official Java site. http://bit.ly/javastringclass

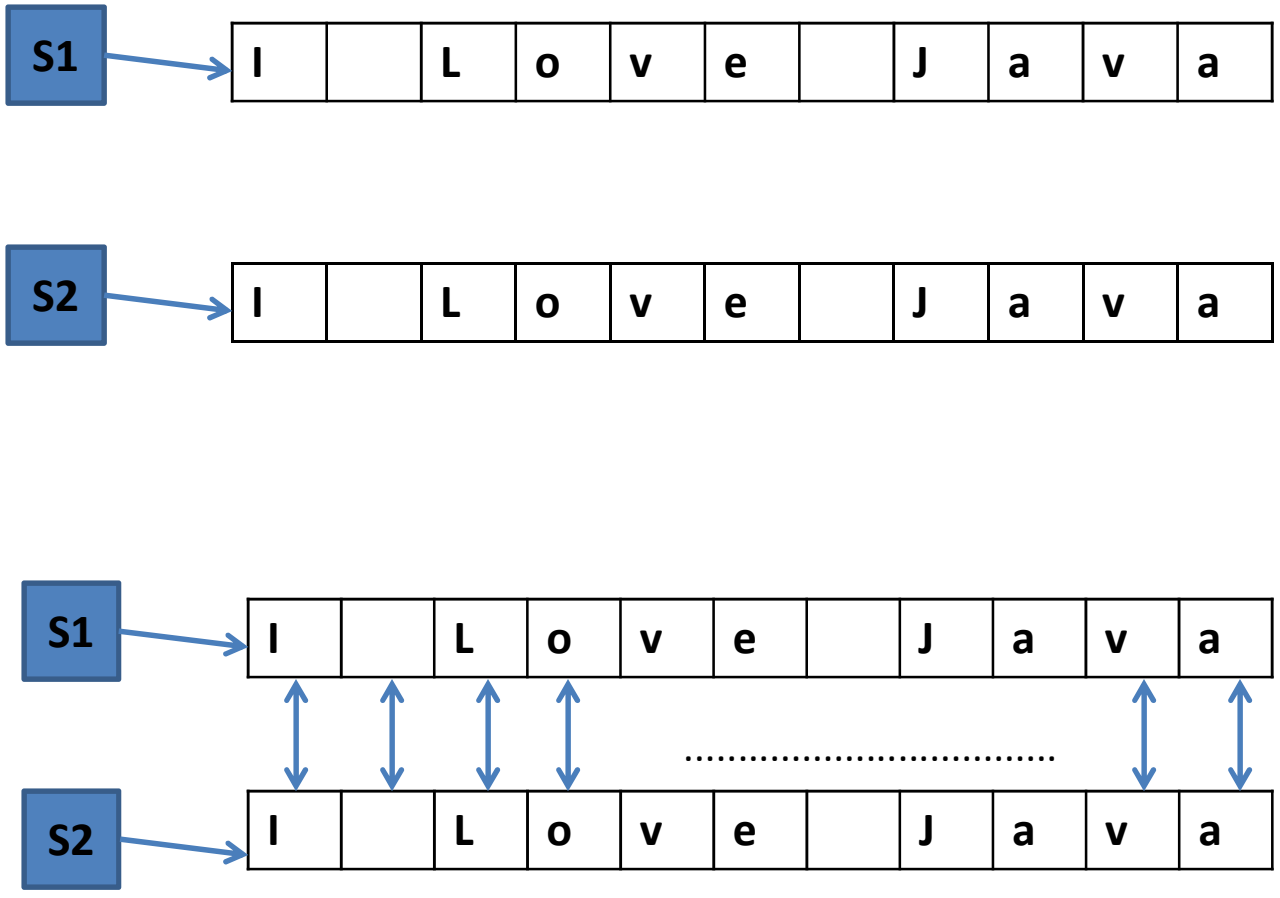
String Equality

```
String s1 = “I Love”;
s1 += “Java”;

String s2 = “I Love”;
s2 += “Java”;

if(s1==s2)   //always false
      //do something

if(s1.equals(s2))  //True
      //do something
```



String Equality

- Normal == method, see if the variables are pointing to the same object or not. Here s1 is pointing to different object and s2 is pointing to some other object, though values in these 2 are exactly same.
- String equals method try to compare the character by character and if values are same then it returns true.
- Here we should note that reference based comparison is very quick and inexpensive but character by character comparison is quite expensive specially in case of very long strings. So in case of very long string comparison, we should take advantage of inexpensive nature of reference based comparison. That's where the intern method comes in.

String Equality & intern Method

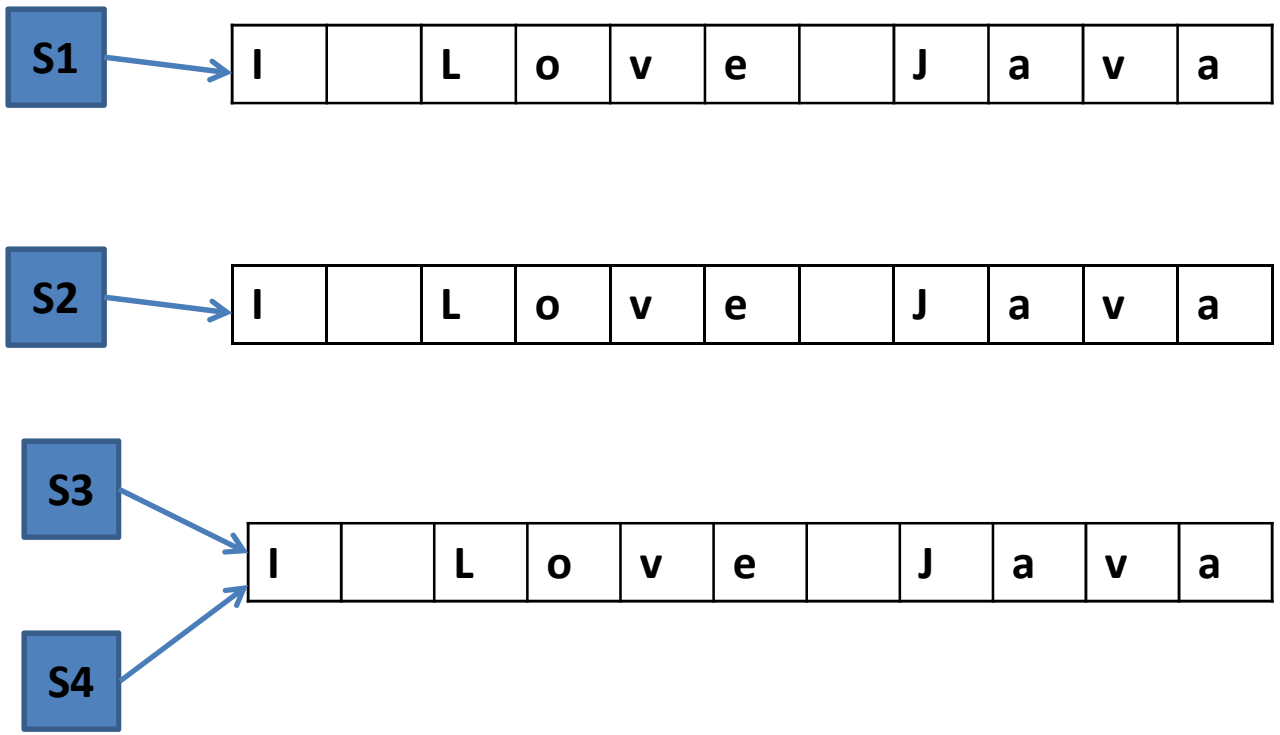
```
String s1 = “I Love”;
s1 += “Java”;

String s2 = “I Love”;
s2 += “Java”;

if(s1==s2)   //always false
      //do something

if(s1.equals(s2))  //True
      //do something

String s3 = s1.intern();
String s4 = s2.intern();
if(s3==s4)   //True
      //do something
```



String Equality & intern Method

- An intern method gives us back a canonicalized reference of a string value.
- It means that, when we will declare another string variable as in case of s3, it will get the same string value as s1. Also, in case of s4, it will get the same string value as s2 but it will not create a new instance. It will 1st check if the string with same exact value exist in the memory block or not and then the intern assures that the 2 string with same value, will reference the exact same object.
- Thus it allows us to do the very inexpensive == comparison by reference. But there is a bit of overhead to interning a string. So the thumb rule is that when String are small then we will use equals method and when there is a series of value in an array or other collection, then we will use intern method to do the comparison.

Converting Non-string Types to Strings

- We often need to convert non-string types into strings having a value of message, which makes it easy to do things like build a message or display output to a user.
  - String.valueOf provides overrides to handle most types.
  - Conversion often happen implicitly in most of the cases. Like primitive type or so.
  - Class conversions controlled by the class toString method.

Converting Non-string Types to Strings

```
int iVal = 100;

String sVal = String.valueOf(iVal);
//gives o/p as “100”

int i = 2, j=3;
int result = i* j;
System.out.println(i+” ”+j+” = ”+result);
/*Fany o/p for better understanding to
      user “2 * 3 = 6” */
```

```
Flight myFlight = new Flight(175);
System.out.println(“My flight is ”+myFlight);
//o/p “My flight is Javaapplication1@74a1448b”
//o/p after below: My Flight is Flight # 175
```

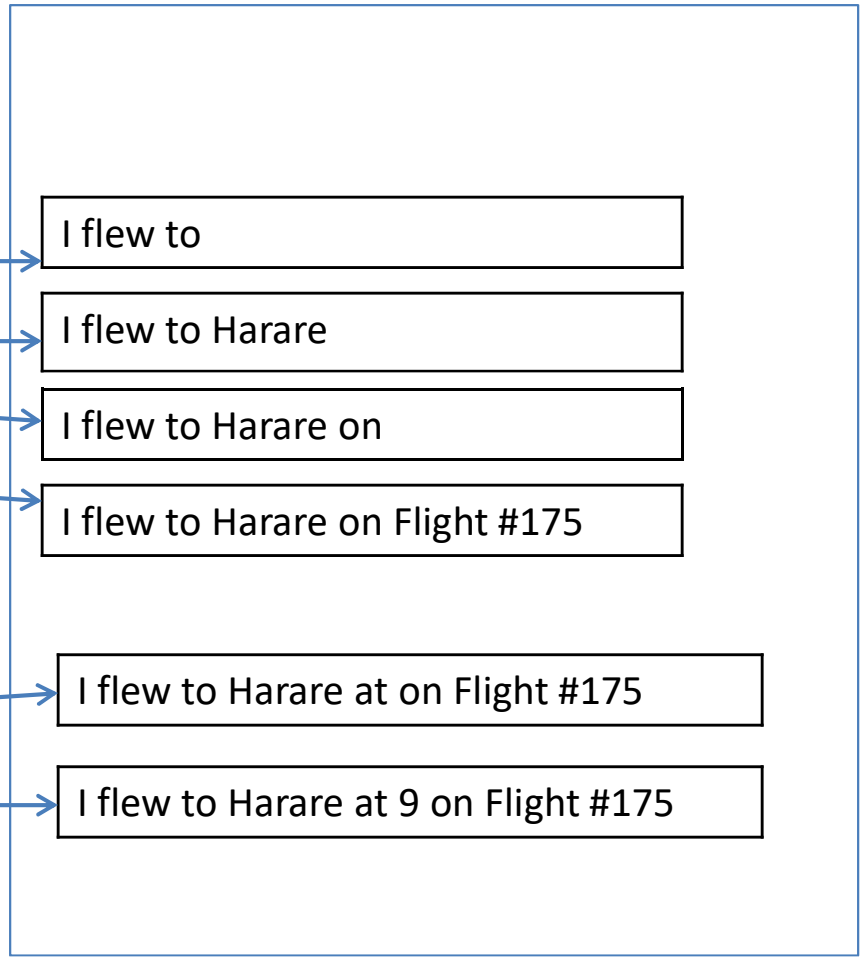
```
Public class Flight{
      int flightNumber;
      char flightClass;
//other method/members elided for clarity
      @override
      public String toString(){
      if(flightNumber > 0)
            return “Flight # ”+flightNumber;
      else
            return “Flight # ”+flightClass;
      }
}
```

StringBuilder

- Since Strings are immutable, that means that any modification of the strings results to a creation of new string and that is not efficient way of programming if it requires so and thats where StringBuilder comes into picture.
- StringBuilder provides mutable string buffer, providing us with an efficient way to manipulate strings.
  - For best performance pre-size buffer.
  - Will grow automatically if needed. But we should minimize this as there is a little bit of overhead each time the system has to grow a StringBuilder instance.
- StringBuilder has number of methods and most common:
  - append : allow us to add new content to the end of StringBuilder
  - insert: allows us to add new content anywhere in StringBuilder

StringBuilder

```
StringBuilder sb = new StringBuilder(40);
Flight myFlight = new Flight(175);
String location = “Harare”;
sb.append(“I flew to ”);
sb.append(“location”);
sb.append(“ on ”);
sb.append(myFlight);
int time = 9;
int pos = sb.lenght() – “ on ”.lenght() –
      myFlight.toString().lenght;
sb.insert(pos, “ at ”);
sb.insert(pos + 4, time);
String message = sb.toString();
```



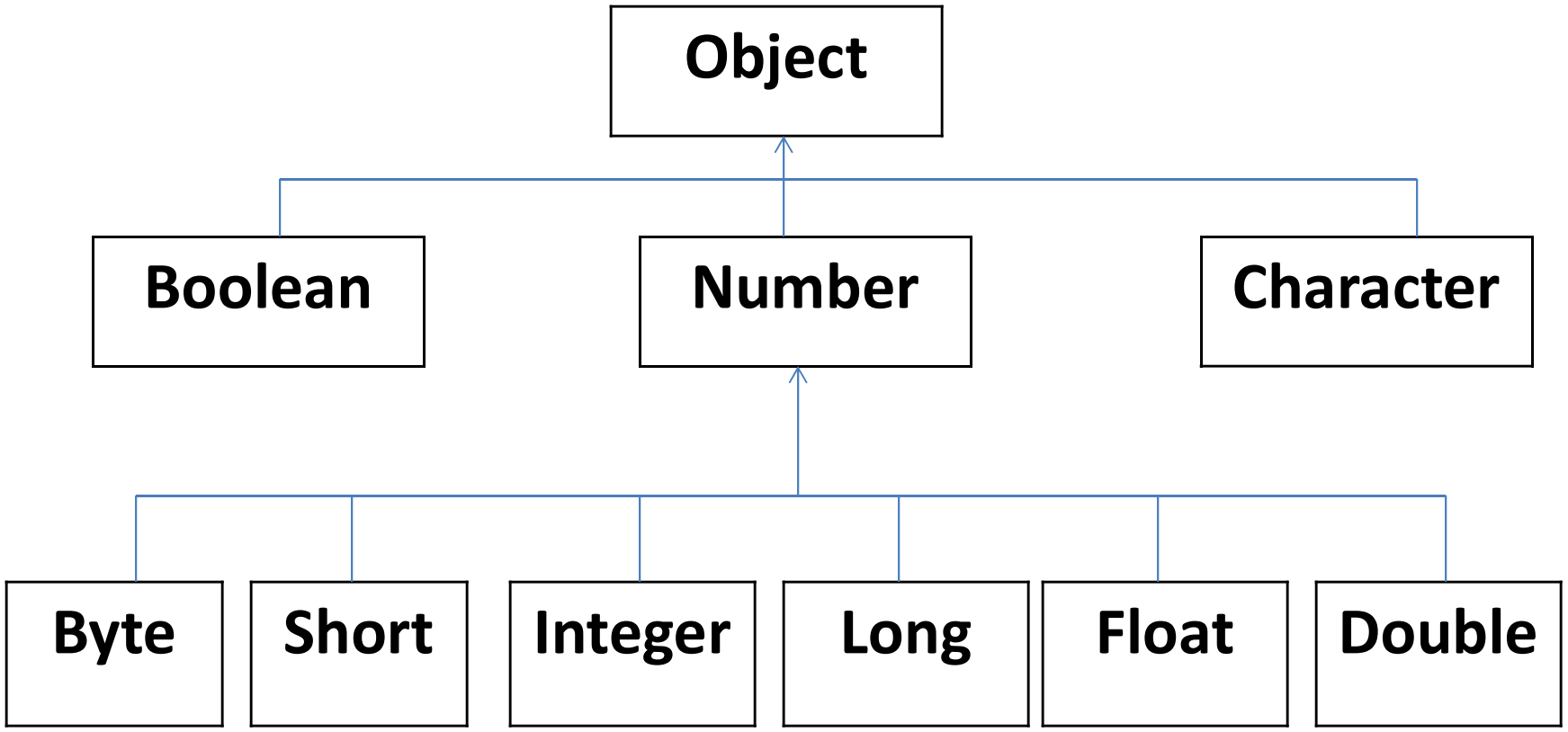
Classes vs Primitives

- Classes provide convenience
  - Common interaction through object class
  - Fields and methods specific to the type
  - Incurs an overhead cost
    - Every class instance has a certain amount of memory that's always taken up just by the fact that it's a class and that's before it even has its own specific values inside of there.
- Primitives provide efficiency
  - Lightweight
  - Cannot expose fields or methods
  - Cannot be treated as object

Primitives Wrapper Classes

- In order to get convenience that we have with classes efficiency as well as lightweight working with primitives type, we can use Primitives Wrapper Classes.
  - Capabilities and overhead of classes
  - Hold primitive values
  - They are indeed classes as they inherit from the object class.
  - Each of these primitive types has a corresponding wrapper class.
  - All wrapper class instances are immutable.

Primitive Wrapper Class Hierarchy



Wrapper Class & Primitive Conversion

- We often have to convert between the Wrapper classes and the primitives. Java provides a number of ways to handle those conversions.
  - Common conversion handled automatically. But there are certain cases where it requires an explicit conversion.
  - Wrapper classes provide methods for explicit conversions.

Wrapper Class & Primitive Conversion

- Primitive to wrapper -> valueOf method. This process of converting primitive to wrapper class is known as boxing, i.e. the idea is that we take a primitive value and wrap it up in a box, which is now the class.
- Wrapper to Primitive -> xxxValue method, where xxx is the name of the primitive type. Thus for Integer we have to write integer.intValue and for double has double.doubleValue. This process of converting Wrapper to primitive is known as unboxing.
- Another type of conversion which Java provides is for String class. For converting string to primitive -> parseXxx (each of the wrapper classes has a parse method on it) i.e. for Integer parse.int It will parse the primitive value and give you back the primitive value corresponding to it.
- For String to primitive -> Java provide an overload of ValueOf that accepts a string and gives us back a reference to a wrapper class.

Wrapper Class & Primitive Conversion

```
Integer a = 100;
int b = a;
Integer c = b;
Integer d= Integer.valueOf(100);
int e = d.intValue();
Integer f = Integer.valueOf(e);
Float g = Float.valueOf(18.125f);
float h = g.floatValue();
String s = “87.44”;
double s1 =
    Double.parseDouble(s);
//This will return the primitive


Double s2 =
    Double.valueOf(s);
//valueOf returns back a reference
    to a wrapper class that has that
    value , 87.44 inside of it.
```

Here I is in capital which means a is a reference to an instance of the integer wrapper class. But 100 is the primitive value 100. Here Java will take care of getting a reference of the integer class that has the value 100 inside it.

When we have declared b equals a then it means that b is a primitive type and ‘a’ is a reference to the wrapper class type. Java will take care of getting the value out of a and just to note that the idea of conversions happen with both variables and literals i.e. we could have integer c equals b, while b is a primitive variable, c is a reference to the integer class. Java will take care of getting a reference to the integer class that has the same value that b has inside of it.

Using Wrapper Classes

- Treat as Object

```
Object[] stuff = new Object[3];
stuff[0] = new Flight();
stuff[1] = new Passenger(0,2);
Stuff[2] = 100;
```

Using Wrapper Classes

- This has a benefit of null references. So we have a distinct idea of a value not being set.

```
public class Flight{

  int flightNumber;
  char flightClass;
  //other method elided for clarity

  @override
  public String toString(){
    if(flightNumber >0)
       return “Flight #” +flightNumber;
    else
      return “Flight Class ” +flightClass;

    }
}
```

```
public class Flight{
  Integer flightNumber;
  Character flightClass;
  //other method elided for clarity

  @override
  public String toString(){
    if(flightNumber != null)
       return “Flight #” +flightNumber;
    else if(flightClass != null)
      return “Flight Class ” +flightClass;
     else
       return “Flight identity not set”;

    }
}
```

Wrapper Class Member

Class	Select Members	Documentation
Byte Short Integer Long	MIN_VALUE, MAX_VALUE, bitCount, toBinaryString	http://bit/javabyte http://bit/javashort http://bit/javainteger http://bit/javalong
Float Double	MIN_VALUE, MAX_VALUE, isInfinite, isNaN	http://bit/javafloat http://bit/javadouble
Character	MIN_VALUE, MAX_VALUE, isDigit, isLetter	http://bit/javacharacter
Boolean	TRUE, FALSE	http://bit/javaboolean

Wrapper Class Equality

```
Integer i1000A= 10*10*10;
Integer i1000B= 100*10;

if(i1000A==i1000B)              //false
//do something

If(i1000A equals(i1000B))       //true
//do something

Integer i8A= 2*2*2;
Integer i8B= 4*2;

if(i8A==i8B)                    //true
//do something
```

Boxing conversions that always returns the same wrapper class instance	
Primitive Type	**Values**
int	-128 to 127
short	-128 to 127
byte	-128 to 127
char	‘\u0000’ to ‘\u00ff’
boolean	True, false

Final Fields

- Sometime in our Program, we will have all fields that we don't allow to be set once they have been initialized. That is why final fields comes in.
- Making a field as final prevents it from being changed once assigned.
  - A simple final field must be set during creation of an object instance.
    - Can be set with field initializer, initialization block, or constructor.

Final Fields

```
public class Passenger{
        private final int freeBags;
        //other member elided for clarity
public Passenger(int freeBags){
        this.freeBags = freeBags;
        }
}


public class Flight{
        static final int  MAX_FLA_SITS = 500; s
        private int eats;
        //other member elided for clarity

public void setSeats(int seats){
        if(seats <= MAXFLA_SEATS)
                this.seats = seats;
        else
                //handle error
        }

}
```

In this passenger class, we may define the business model as, a number of free bags is an aspect of the passenger. So it can only be set during the creation of the passenger. We can do so by including the word final as shown in the program. By Including the word final, any attempt to set free bags after it's been set in the constructor would actually result in a compile error. So compiler enforces the rule that it can't be assigned once it's been set.

In Java there is also static final field. Adding a static modifier make a final field a named constant. A static final field can't be changed in any object instance. Its value is tied to the class itself. For example Zimbabwe government has rule that a flight can't exceed a maximum of 500 seats.

Enumeration Types

- Enumeration types useful for defining a type with a finite list of valid values.
  - Declare with keyword enum
  - We can assign to any instance using name_of_enum.values_declared_in_list_enum
- We will see an example for enumeration to make it more clear.

Enumeration Types

```
public enum FlightCrewJob{
    Pilot,
    CoPilot,
    FlightAttendant,
    AirMarshal1
}
```

```
public class CrewMember{
    private FlightCrewJob job;
    //other member elided for clarity

    public CrewMember(FlightCrewJob job){
        this.job = job;
    }
    public void setJob(FlightCrewJob job){
        this.job = job';
    }
}
```

```
CrewMember judy = new
    CrewMember(FlightCrewJob.CoPilot);


//...Judy got Promotion
Judy.setJob(FlightCrewJob.Pilot);
```

This is just a basic idea of enumeration type. It has lot of advanced capabilities beyond this.

Summary

- String class stores an immutable sequence of Unicode characters
  - Implement toString method to provide conversion to a string.
- StringBuilder class provide an efficient way of manipulate string values.
- Primitive wrapper classes bring class capabilities to primitive values.
  - Wrapper classes much less efficient than primitive types.
- Final field prevent a value from being changed once assigned.
  - Simple final fields must be set during object instance creation
  - Static final fields act as named constant
- Enumeration types useful for defining a type with a finite list of values.