Enumeration, Autoboxing

Enumerations

- Enumerations are available in JDK 5 and higher versions.
- An *enumeration* is a list of named constants.
- Though, for a first look, they seem same as enumerations in C/C++, actually in Java, they are class types.
- That is, in Java, enumerations can have constructors, methods and variables.
- An enumeration is created using the keyword ***enum.***
- For example,

  enum Person

  {Married, Unmarried, Divorced, Widowed}
- The identifiers like ***Married, Unmarried*** etc. are called as ***enumeration Constants.***
- Each such constant is implicitly considered as a ***public static final*** member of Person.

- The type of enumeration constants is the type of the enumeration in which they are declared, which is **Person** in this case.
- Thus, in the language of Java, these constants are called ***self-typed***, in which "self" refers to the enclosing enumeration.
- After defining enumeration, we can create a variable of that type.
- Though enumeration is a class type, we need not use ***new*** keyword for variable creation, rather we can declare it just like any primitive data type.
- For example,
  Person p= Person.Married;
- We can use == operator for comparing two enumeration variables.
- They can be used in ***switch-case*** also.
- Printing an enumeration variable will print the constant name. That is,
  System.out.println(p);
  will print Married.


```
enum Person
{    Married, Unmarried, Divorced, Widowed}

class EnumDemo
{    public static void main(String args[])
     {       Person p1;
             p1=Person.Unmarried;
             System.out.println("Value of p1 :" + p1);
             Person p2= Person.Widowed;
             if(p1==p2)
                        System.out.println("p1 and p2 are same");
             else
                        System.out.println("p1 and p2 are different");
             switch(p1)
             {          case Married: System.out.println("p1 is Married");
                                      break;
                        case Unmarried: System.out.println("p1 is Unmarried");
                                        break;
                        case Divorced: System.out.println("p1 is Divorced");
                                       break;
                        case Widowed: System.out.println("p1 is Widowed");
                                      break;
             }
     }
}
```

The values( ) and valueOf( ) Methods

- All enumerations automatically contain two predefined methods: **values( )** and **valueOf( )**.

- Their general forms are shown here:

  public static *enum-type*[ ] values( )

  public static *enum-type* valueOf(String *str*)

- The **values( )** method returns an array that contains a list of the enumeration constants.

- The**valueOf( )** method returns the enumeration constant whose value corresponds to the string passed in *str.*

- In both cases, *enum-type* is the type of the enumeration.


```
enum Person
{    Married, Unmarried, Divorced, Widowed }

class EnumDemo
{
    public static void main(String args[])
    {
            Person p;

            System.out.println("Following are Person constants:");
            Person all[]=Person.values();
            for(Person p1:all)
                    System.out.println(p1);

            p=Person.valueOf("Married");
            System.out.println("p contains "+p);
    }
}
```

Output:

```
Following are Person constants:
Married
Unmarried
Divorced
Widowed
p contains Married
```

Java Enumerations Are Class Types

- Java enumeration is a class type.
- We can give them constructors, add instance variables and methods, and even implement interfaces.
- It is important to understand that each enumeration constant is an object of its enumeration type.
- Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created.
- Also, each enumeration constant has its own copy of any instance variables defined by the enumeration.


```
enum Apple
{
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
    private int price;

    Apple(int p)
    {       price = p; }
    int getPrice()
    {       return price;  }
}
class EnumDemo
{
    public static void main(String args[])
    {
            Apple ap;
            System.out.println("Winesap costs " + Apple.Winesap.getPrice() + " cents.\n");

            System.out.println("All apple prices:");
            for(Apple a : Apple.values())
                    System.out.println(a + " costs " + a.getPrice() + " cents.");
    }
}
```

Output:

```
Winesap costs 15 cents.
All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

- Here, we have member variable ***price,*** a constructor and a member method.

- When the variable **ap** is declared in **main( )**, the constructor for **Apple** is called once for each constant that is specified.

- Although the preceding example contains only one constructor, an **enum** can offer two or more overloaded forms, just as can any other class.

- Two restrictions that apply to enumerations:
  - an enumeration can't inherit another class.
  - an **enum** cannot be a superclass.

Enumerations Inherit Enum

- All enumerations automatically inherit one: **java.lang.Enum**.

- This class defines several methods that are available for use by all enumerations.

- We can obtain a value that indicates an enumeration constant's position in the list of constants.

- This is called its *ordinal value,* and it is retrieved by calling the **ordinal( )** method, shown here:

  final int ordinal( )

- It returns the ordinal value of the invoking constant.

- **Ordinal values begin at zero.**

- We can compare the ordinal value of two constants of the same enumeration by using the **compareTo( )** method.

- It has this general form:

  final int compareTo(*enum-type e*)

- The usage will be –

  e1.compareTo(e2);

- Here, e1 and e2 should be the enumeration constants belonging to same enum type.

- If the ordinal value of e1 is less than that of e2, then compareTo() will return a negative value.

- If two ordinal values are equal, the method will return zero.

- Otherwise, it will return a positive number.

- We can compare for equality an enumeration constant with any other object by using **equals( )**, which overrides the **equals( )** method defined by **Object**.


```
enum Person
{    Married, Unmarried, Divorced, Widowed }
enum MStatus
{    Married, Divorced  }
class EnumDemo
{
    public static void main(String args[])
    {
      Person p1, p2, p3;
      MStatus m=MStatus.Married;

      System.out.println("Ordinal values are: ");
      for(Person p:Person.values())
            System.out.println(p + " has a value " +
                                          p.ordinal());

      p1=Person.Married;
      p2=Person.Divorced;
      p3=Person.Married;

       if(p1.compareTo(p2)<0)
            System.out.println(p1 + " comes before "+p2);
       else if(p1.compareTo(p2)==0)
            System.out.println(p1 + " is same as "+p2);
       else
            System.out.println(p1 + " comes after "+p2);

if(p1.equals(p3))
  System.out.println("p1 & p3 are same");
 if(p1==p3)
   System.out.println("p1 & p3 are same");

if(p1.equals(m))
  System.out.println("p1 & m are same");
else
   System.out.println("p1 & m are not
same");

//if(p1==m)
//System.out.println("p1 & m are same");
            }
}
```

Type Wrappers

- Java uses primitive types (also called simple types), such as **int** or **double**, to hold the basic data types supported by the language.
- Primitive types, rather than objects, are used for these quantities for the sake of performance.
- Using objects for these values would add an unacceptable overhead to even the simplest of calculations.
- Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**.
- Despite the performance benefit offered by the primitive types, there are times when you will need an object representation.
- For example, **you can't pass a primitive type by reference to a method.**
- Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types.
- To handle these (and other) situations, Java provides ***type wrappers***, which are classes that encapsulate a primitive type within an object.

- The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**.
- These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.
- The process of encapsulating a value within an object is called ***boxing***.
- The process of extracting a value from a type wrapper is called ***unboxing***.

Primitive	Wrapper
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	java.lang.Void

Character

- **Character** is a wrapper around a **char**.
- The constructor for **Character** is
  Character(char *ch*)
- Here, *ch* specifies the character that will be wrapped by the **Character** object being created.
- To obtain the **char** value contained in a **Character** object, call **charValue()**, shown here:
  char charValue( )
- It returns the encapsulated character.

Boolean

- **Boolean** is a wrapper around **boolean** values.
- It defines these constructors:
  Boolean(boolean *boolValue*)
  Boolean(String *boolString*)
- In the first version, *boolValue* must be either **true** or **false**.
- In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.
- To obtain a **boolean** value from a **Boolean** object, use
  boolean booleanValue( )
- It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

- The most commonly used type wrappers are those that represent numeric values.
- All of the numeric type wrappers inherit the abstract class **Number**.

- **Number** declares methods that return the value of an object in each of the different number formats.

- These methods are shown here:

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

- For example, **doubleValue( )** returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on.

- These methods are implemented by each of the numeric type wrappers.

- All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value.

- For example, here are the constructors defined for **Integer**:

```
Integer(int num)
Integer(String str)
```

- If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.

- All of the type wrappers override **toString( )**.

- It returns the human-readable form of the value contained within the wrapper.

- This allows you to output the value by passing a type wrapper object to **println( )**, for example, without having to convert it into its primitive type.

```
class TypeWrap
{
    public static void main(String args[])
    {
        Character ch=new Character('#');
        System.out.println("Character is " + ch.charValue());

        Boolean b=new Boolean(true);
        System.out.println("Boolean is " + b.booleanValue());

        Boolean b1=new Boolean("false");
        System.out.println("Boolean is " + b1.booleanValue());

        Integer iOb=new Integer(12);          //boxing
        int i=iOb.intValue();                 //unboxing
        System.out.println(i + " is same as " + iOb);

        Integer a=new Integer("21");
        int x=a.intValue();
        System.out.println("x is " + x);

        String s=Integer.toString(25);
        System.out.println("s is " +s);
    }
}
```

Output:

```
Character is #
Boolean is true
Boolean is false
12 is same as 12
x is 21
s is 25
```


Autoboxing

- In older versions of Java, the programmer was supposed to do boxing and unboxing.
- Beginning with JDK 5, Java added the features: ***autoboxing*** and ***auto-unboxing.***
- **Autoboxing** is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.
- There is no need to explicitly construct an object.
- **Auto-unboxing** is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.
- There is no need to call a method such as **intValue( )** or **doubleValue( )**.

- The addition of autoboxing and auto-unboxing greatly streamlines the coding of several algorithms, removing the tedium of manually boxing and unboxing values.
- It also helps prevent errors.
- Moreover, it is very important to generics, which operates only on objects.
- Finally, autoboxing makes working with the Collections Framework much easier.
- With autoboxing it is no longer necessary to manually construct an object in order to wrap a primitive type.
- You need only assign that value to a type-wrapper reference.
- Java automatically constructs the object for you.
- For example, here is the modern way to construct an **Integer** object that has the value 100:

  Integer iOb = 100; // autobox an int
- To unbox an object, simply assign that object reference to a primitive-type variable.

  int i = iOb; // auto-unbox

Autoboxing and Methods

- Autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method.
- autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type.

```
class AutoBox
{
    static int m(Integer v)
    {
      return v ;
    }

    public static void main(String args[])
    {
      Integer iOb = m(100);
      System.out.println(iOb);
    }
}
```

Autoboxing/Unboxing Occurs in Expressions

- Within an expression, a numeric object is automatically unboxed.
- The outcome of the expression is reboxed, if necessary.


```
class AutoBox
{   public static void main(String args[])
    {       Integer iOb=100, iOb2;
            int i;
            Double dOb=21.3;

            System.out.println("Original value of iOb: " + iOb);
            ++iOb;              //auto-unboxing to int type then result is boxed back to Integer
            System.out.println("After ++iOb: " + iOb);

            iOb2 = iOb + (iOb / 3);         //auto-unboxing and then re-boxing
            System.out.println("iOb2 after expression: " + iOb2);

            i = iOb + (iOb / 3);    //auto-unboxing but not re-boxing as LHS is of int type
            System.out.println("i after expression: " + i);

            dOb=dOb+iOb;
            System.out.println("double value is " + dOb);
            switch(iOb)
            {           case 100: System.out.println("Hundred"); break;
                        case 101: System.out.println("Hundred One"); break;
                        default: System.out.println("Error"); break;
            }
    }
}
```

Autoboxing/Unboxing Boolean and Character Values:

```
class AutoBox
{
        public static void main(String args[])
        {
                Boolean b = true;
                if(b)       //un-boxed
                        System.out.println("b is true");

                Character ch = 'x';          // box a char
                char ch2 = ch;   // unbox a char
                System.out.println("ch2 is " + ch2);
        }
}
```

- Though we say that *if, while, do-while* requires *boolean* data type, we can achieve the same result with *Boolean object* also as Java does auto-unboxing.

Autoboxing/Unboxing Helps Prevent Errors:

- Autoboxing always creates the proper object, and auto-unboxing always produces the proper value.

- So, there is no way for the process to produce the wrong type of object or value.

```
class AutoBox
{
        public static void main(String args[])
        {
                Integer iOb=500;
                int x=iOb ;
                int y=iOb.byteValue();
                System.out.println(x + “  “ + y);      // prints 500   -12
        }
}
```

- In the rare instances where you want a type different than that produced by the automated process, you can still manually box and unbox values.

A Word of Warning:

- Since Java includes autoboxing and auto-unboxing, some might be tempted to use objects such as **Integer** or **Double** exclusively, ignoring primitives altogether.

- For example, with autoboxing/unboxing it is possible to write code like this:

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

- Although this code is technically correct and work properly, it is a very bad use of autoboxing/unboxing.

- It is far less efficient than the equivalent code written using the primitive type **double**.

- The reason is that each autobox and auto-unbox adds overhead that is not present if the primitive type is used.

- In general, you should restrict your use of the type wrappers to only those cases in which an object representation of a primitive type is required.

Generics

- Introduction of generics to JDK5 has changed Java in two important ways:
  - it added a new syntactical element to the language.
  - it caused changes to many of the classes and methods in the core API.

- Through the use of generics, it is possible to create classes, interfaces, and methods that will work in a type-safe manner with various kinds of data.

- Many algorithms are logically the same irrespective of the data type on which they are applied.

- For example, the underlying mechanism for stack is same whether it is storing int, String etc.

- With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types without any additional effort.

What are Generics?

- The term *generics* means ***parameterized types***.
- Parameterized types enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.
- Using generics, it is possible to create a single class (or method or interface) that automatically works with different types of data.
- A class (or interface or method) that operates on a parameterized type is called ***generic class (generic interface or generic method).***
- In older versions of Java, the ***Object*** class references were used to operate upon any type of data to partially achieve the behavior of generics.
- But when ***Objects*** are used, we were supposed to explicit casting each time when a particular data type was needed.
- With generics, all casts are automatic and implicit thus allowing safe coding.

The general form of Generic class

- The syntax of declaring a generic class is –

  class *class-name<type-param-list>*

  {

  //body of class

  }

- Here ***type-param-list*** is a list of ***type parameters (or place-holders).***
- To declare a reference (or object) to a generic class, the syntax is –

  class-name<type-arg-list> var-name =

  new *class-name<type-arg-list>(cons-arg-list)*;

- Here ***type-arg-list*** are list of ***type arguments*** that are passed to the corresponding ***type parameters*** in ***type-param-list***.
- *cons-arg-list* is a list of arguments to be passed to the class constructor

A simple Generics Example

```
class Gen<T>
{
    T ob;
    Gen(T o)
    {
        ob = o;
    }

    T getob()
    {
        return ob;
    }

    void showType()
    {
        System.out.println("Type of T is " + ob.getClass().getName());
    }
}
```



```
class GenDemo
{
    public static void main(String args[])
    {

        Gen<Integer> iOb;
        iOb = new Gen<Integer>(88);

        iOb.showType();

        int v = iOb.getob();
        System.out.println("value: " + v);

        Gen<String> strOb = new Gen<String>("Generics Test");
        strOb.showType();

        String str = strOb.getob();
        System.out.println("value: " + str);
    }
}
```

- The **showType( )** method displays the type of **T** by calling **getName( )** on the **Class** object returned by the call to **getClass( )** on **ob**.
- The **getClass( )** method is defined by **Object** and is thus a member of all class types.
- **Class** defines the **getName( )** method, which returns a string representation of the class name.
- Java compiler does not actually create different versions of any other generic class.
- Instead, the compiler removes all generic type information, substituting the necessary casts, to make your code ***behave as if*** a specific version of **Gen** were created.
- **Thus, there is really only one version of Gen that actually exists in your program (This is not the case with C++ templates).**
- The process of removing generic type information is called ***erasure.***

- The type-checking mechanism of Java ensures type safety.
- That is, following **is an error.**

```
Gen<Integer> iOb;
iOb=new Gen<Double>(3.5);
```

- The statement

  Gen<Integer> iOb =new Gen<Integer>(88);

  makes use of autoboxing to encapsulate the **int** value 88 to **Integer** type.

Generics work with only Objects:

- When declaring an instance of a generic type, the type argument passed to the type parameter must be a class type.

- You cannot use a primitive type, such as **int** or **char**.

- For example,

```
Gen<int> iOb=new Gen<int>(88); //error!!
```


Generic Types Differ Based on Their Type Arguments:

- A reference of one specific version of a generic type is not type compatible with another version of the same generic type.

- For example,

```
Gen<Integer>iOb;
Gen<String>strOb;
iOb = strOb;  // Error!!
```

- Even though both **iOb** and **strOb** are of type **Gen<T>**, they are references to different types because their type parameters differ.

- This is part of the way that generics add type safety and prevent errors.

How Generics Improve Type Safety:

- Since ***Object*** is a base class for all other classes, it can be used as a container (place holder) for any other class.
- Thus, generic nature can be obtained by using ***Object*** and without using actual generic types.
- This can be achieved just by specifying *Object* as the data type and using proper type-casts.
- But still, generic types are better way to use, since they automatically ensure the type-safety for all operations involved in generic implementation.


```
class NonGen
{    Object ob;

     NonGen(Object o)
     {
            ob = o;
     }

     Object getob()
     {
             return ob;
     }

     void showType()
     {
            System.out.println("Type of ob is "
              + ob.getClass().getName());
     }
}
```

```
class NonGenDemo
{    public static void main(String args[])
     {
        NonGen iOb;
        iOb = new NonGen(88);
        iOb.showType();

        int v = (Integer)iOb.getob();  //note casting
        System.out.println("value: " + v);

        NonGen strOb = new NonGen("Test");
        strOb.showType();

        String str = (String) strOb.getob(); //casting
        System.out.println("value: " + str);

        iOb = strOb;
        v = (Integer) iOb.getob();   // run-time error!
     }
}
```

Output:

```
Type of ob is java.lang.Integer
value: 88
Type of ob is java.lang.String
value: Non-Generics Test
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot
 be cast to java.lang.Integer
      at NonGenDemo.main(NonGenDemo.java:40)
```

A Generic Class with Two Type Parameters

- You can declare more than one type parameter in a generic type.
- To specify two or more type parameters, simply use a comma-separated list.


```
class TwoGen<T, V>
{   T ob1;
    V ob2;

    TwoGen(T o1, V o2)
    {
        ob1 = o1;
        ob2 = o2;
    }

    T getob1()
    {
        return ob1;
    }

    V getob2()
    {
        return ob2;
    }
    void showTypes()
    {
        System.out.println("Type of T is " + ob1.getClass().getName());
        System.out.println("Type of V is " + ob2.getClass().getName());
    }
}
```

```
class SimpGen
{  public static void main(String args[])
   {
     TwoGen<Integer, String> t = new  TwoGen<Intege
String>(88, "Generics");

     t.showTypes();
     int v = t.getob1();
     System.out.println("value: " + v);

     String str = t.getob2();
     System.out.println("value: " + str);

     TwoGen<String, Double> t1=new
          TwoGen<String, Double>("Hello", 21.3);

      t1.showTypes();
   }
}
```

String Handling

- A string is a sequence of character and Java implements strings as objects of type ***String.***
- Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient.
- For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.
- Also, **String** objects can be constructed a number of ways, making it easy to obtain a string when needed.
- Once a **String** object has been created, you cannot change the characters of that string.
- Whenever we need any modifications, a new string object containing modifications has to be created.
- However, a variable declared as String reference can point to some other String object, and hence can be changed.

- In case, we need a modifiable string, we should use ***StringBuffer*** or ***StringBuilder*** classes.
- ***String, StringBuffer*** and ***StringBuilder*** classes are in ***java.lang*** and are ***final*** classes.
- Thus, no class can inherit these classes.
- All these classes implement ***CharSequence*** interface.

The String Constructors

There are several constructors for String class.

- To create an empty string, use default constructor:
  String s= new String();

2. To create a string and initialize:
   String s= new String("Hello");

3. To create a string object that contains same characters as another string object:
   String(String strObj);

   For example,
   String s= new String("Hello");
   String s1= new String(s);

4. To create a string having initial values:
   String(char *chars[]*)

   For example,
   char ch[]={'h', 'e', 'l', 'l', 'o'};
   String s= new String(ch); //s contains hello

5. To specify a subrange of a character array as an initializer use the following constructor:
   String(char *chars*[ ], int *startIndex*, int *numChars*)

   For example,
   char ch[]={'a', 'b', 'c', 'd', 'e', 'f', 'g'};
   String s= new String(ch, 2, 3); //s contains cde

- Even though Java's **char** type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.

- Because 8-bit ASCII strings are common, the **String** class provides constructors that initialize a string when given a **byte** array.

6. The general forms are:

    String(byte *asciiChars*[ ])
    String(byte *asciiChars*[ ], int *startIndex*, int *numChars*)

    For example,

```
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);      // s1 contains ABCDEF
String s2 = new String(ascii, 2, 3);          // s2 contains CDE
```


- JDK 5 and higher versions have two more constructors.

- The first one supports the extended Unicode character set.

7. The general form:

    String(int *codePoints*[ ], int *startIndex*, int *numChars*)

    here, *codePoints* is array containing Unicode points.

8. Another constructor supports ***StringBuilder:***

    String(StringBuilder *strBuildObj*)

String Length:

- The length of a string is the number of characters that it contains.

- To obtain this value, call the **length( )** method, shown here:

  int length( )

- For example,

```
String s=new String("Hello");
System.out.println(s.length());      //prints 5
```


Special String Operations

- Java supports many string operations.

- Though there are several string handling methods are available, for the use of programmer, Java does many operations automatically without requiring a call for separate method.

- This adds clarity to the program.

- We will now see few of such operations.

String Literals:

- Instead of using character arrays and *new* operator for creating string instance, we can use string literal directly.

- For example,

```
char ch[]={'H', 'e', 'l', 'l', 'o'};
String s1=new String(ch);
```

or

```
String s2= new String ("Hello");
```

Can be re-written, for simplicity, as –

```
String s3="Hello";    //usage of string literal
```

- A ***String*** object will be created for every string literal and hence, we can even use,

```
System.out.println("Hello".length());        //prints 5
```


String Concatenation:

- Java does not allow any other operator than + on strings.

- Concatenation of two or more String objects can be achieved using + operator.

- For example,

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);        //prints He is 9 years old.
```

- One practical use of string concatenation is found when you are creating very long strings.

- Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the **+** to concatenate them.

```
String longStr = "This could have been " +
                 "a very long line that would have " +
                 "wrapped around. But string concatenation " +
                 "prevents this.";
System.out.println(longStr);
```

String Concatenation with Other Data types:

- We can concatenate String with other data types.

- For example,

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);          //prints He is 9 years old.
```

- Here, the **int** value in **age** is automatically converted into its string representation within a **String** object.

- The compiler will convert an operand to its string equivalent whenever the other operand of the **+** is an instance of **String**.

- But, we should be careful while mixing data types:

```
String s= “Four : ” + 2 + 2;
System.out.println(s);          //prints Four : 22
```

- This is because, “Four :” is concatenated with 2 first, then the resulting string is again concatenated with 2.

- We can prevent this by using brackets:

```
String s = “Four : ” + (2+2);
System.out.println(s);          //prints Four : 4
```


String Conversion and toString:

- Java uses **valueOf( )** method for converting data into its string representation during concatenation.

- **valueOf( )** is a string conversion method defined by **String**.

- **valueOf( )** is overloaded for all the primitive types and for type **Object**.

- For the primitive types, **valueOf( )** returns a string that contains the human-readable equivalent of the value with which it is called.

- For objects, **valueOf( )** calls the **toString( )** method on the object.

- Every class implements **toString( )** because it is defined by **Object**.
- However, the default implementation of **toString( )** is seldom sufficient.
- For our own classes, we may need to override toString() to give our own string representation for user-defined class objects.
- The **toString( )** method has this general form:

  String toString( )
- To implement **toString( )**, simply return a **String** object that contains the human-readable string that appropriately describes an object of our class.


```
class Box
{   double width, height, depth;

    Box(double w, double h, double d)
    {       width = w;
            height = h;
            depth = d;
    }
    public String toString()
    {
            return "Dimensions are " + width + " by " + depth + " by " + height + ".";
    }
}
class StringDemo
{
    public static void main(String args[])
    {
            Box b = new Box(10, 12, 14);
            String s = "Box b: " + b;                   // concatenate Box object
            System.out.println(s);                      // convert Box to string
            System.out.println(b);
    }
}
```

Output:

```
Box b: Dimensions are 10.0 by 14.0 by 12.0
Dimensions are 10.0 by 14.0 by 12.0
```

Note: Observe that, **Box**'s **toString( )** method is automatically invoked when a **Box** object is used in a concatenation expression or in a call to **println( )**.

Character Extraction

- The ***String*** class provides different ways for extracting characters from a string object.

- Though a ***String*** object is not a character array, many of the ***String*** methods use an index into a string object for their operation.

charAt() :

- This method is used to extract a single character from a **String.**
- It has this general form:

  char charAt(int *where*)
- Here, *where* is the index of the character that you want to obtain.

- The value of *where* must be nonnegative and specify a location within the string.

```
char ch;
ch= “Hello”.charAt(1);        //ch now contains e
```


getChars() :

- If you need to extract more than one character at a time, you can use the **getChars( )** method.

- It has this general form:

  void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

sourceStart	specifies the index of the beginning of the substring
sourceEnd	specifies an index that is one past the end of the desired substring. (i.e. the substring contains the characters from *sourceStart* through *sourceEnd*–1)
target	specifies the array which receives the substring
targetStart	is the index within *target* at which the substring will be copied

- Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

```
class StringDemo1
{
    public static void main(String args[])
    {
        String s = "This is a demo of the getChars method.";
        int start = 10;
        int end = 14;
        char buf[] = new char[end - start];
        s.getChars(start, end, buf, 0);
        System.out.println(buf);
    }
}
```

Output:

demo

getBytes() :

- **getBytes()** is an alternative to **getChars( )** that stores the characters in an array of bytes.
- It uses the default character-to-byte conversions provided by the platform.
- Here is its simplest form:

  byte[ ] getBytes( )
- Other forms of **getBytes( )** are also available.
- **getBytes( )** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters.
- For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray() :

- If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**.

- It returns an array of characters for the entire string. It has this general form:

  char[ ] toCharArray( )

- This function is provided as a convenience, since it is possible to use **getChars( )** to achieve the same result.

String Comparison

- The **String** class provides several methods to compare strings or substrings within strings.

equals( ) and equalsIgnoreCase( )

- To compare two strings for equality, we have tow methods:

  boolean equals(Object *str*)

  boolean equalsIgnoreCase(String *str*)

- Here, *str* is the **String** object being compared with the invoking **String** object.

- The first method is case sensitive and returns true, if two strings are equal.

- The second method returns true if two strings are same, whatever may be their case.

```
class equalsDemo
{
    public static void main(String args[])
    {
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = "Good-bye";
        String s4 = "HELLO";

        System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
        System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));
        System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));
        System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
                                                    s1.equalsIgnoreCase(s4));
    }
    }
```

Output :

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```


regionMatches( )

- The **regionMatches( )** method compares a specific region inside a string with another specific region in another string.
- There is an overloaded form that allows you to ignore case in such comparisons.
- Here are the general forms for these two methods:
  boolean regionMatches(int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)

  boolean regionMatches(boolean *ignoreCase*, int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)

startIndex	specifies the index at which the region begins within the invoking **String** object.
str2	the **String** being compared.
str2StartIndex	The index at which the comparison will start within *str2*.
numChars	The length of the substring being compared.
ignoreCase	used in second version. If it is **true**, the case of the characters is ignored. Otherwise, case is significant.

```
class StringDemo
{
   public static void main(String args[])
   {
        String s1= "Hello How are you?";
        String s2= "how";

        System.out.println(s1.regionMatches(6,s2,0,3));
        System.out.println(s1.regionMatches(true,6,s2,0,3));
   }
}
```

Output:

```
false
true
```


startsWith( ) and endsWith( ):

- These are the specialized versions of he **regionMatches( )** method.
- The **startsWith( )** method determines whether a given **String** begins with a specified string. The, **endsWith( )** determines whether the **String** in question ends with a specified string.
- They have the following general forms:
  boolean startsWith(String *str*)
  boolean endsWith(String *str*)
- For example,
  "Foobar".endsWith("bar") //true
  "Foobar".startsWith("Foo") //ture
- A second form of **startsWith( )**, lets you specify a starting point:
  boolean startsWith(String *str*, int *startIndex*)
- Here, *startIndex* specifies the index into the invoking string at which point the search will begin.
- For example, "Foobar".startsWith("bar", 3) returns true.

equals( ) Versus ==:

- The **equals( )** method compares the characters inside a **String** object.
- The **==** operator compares two object references to see whether they refer to the same instance.

```
class Demo
{
    public static void main(String args[])
    {
      String s1 = "Hello";
      String s2 = new String(s1);
      System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
      System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
    }
 }
```

Output:

```
true
false
```


compareTo():

- This method is used to check whether a string is *less than*, *greater than* or *equal to* the other string.
- The meaning of *less than*, *greater than* refers to the dictionary order (based on Unicode).
- It has this general form:

  int compareTo(String *str*)

- If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( )**, as shown here:

  int compareToIgnoreCase(String *str*)

```
class SortString
{
    static String arr[] = {"hello", "How", "are", "You?"};

    public static void main(String args[])
    {
        for(int j = 0; j < arr.length; j++)
        {
            for(int i = j + 1; i < arr.length; i++)
                if(arr[i].compareTo(arr[j]) < 0)
                {
                    String t = arr[j];
                    arr[j] = arr[i];
                    arr[i] = t;
                }
            System.out.println(arr[j]);
        }
    }
}
```

Output:

How
You?
are
hello

Searching Strings

The String class provides two different overloaded methods that allow you to search a string for a specified character or substring.

Method	Purpose
int indexOf(int *ch*)	To search for the first occurrence of a character
int lastIndexOf(int *ch*)	To search for the last occurrence of a character,
int indexOf(String *str*)	To search for the first or last occurrence of a substring
int lastIndexOf(String *str*)	
int indexOf(int *ch*, int *startIndex*)	Used to specify a starting point for the search. Here, *startIndex* specifies the index at which point the search begins. For **indexOf( )**, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )**, the search runs from *startIndex* to zero.
int lastIndexOf(int *ch*, int *startIndex*)	
int indexOf(String *str*, int *startIndex*)	
int lastIndexOf(String *str*, int *startIndex*)	

```
class Demo
{   public static void main(String args[])
    {       String s = "Now is the time for all good men to come to the aid of their country.";

            System.out.println(s);
            System.out.println("indexOf(t) = " + s.indexOf('t'));
            System.out.println("lastIndexOf(t) = " + s.lastIndexOf('t'));
            System.out.println("indexOf(the) = " + s.indexOf("the"));
            System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
            System.out.println("indexOf(t, 10) = " + s.indexOf('t', 10));
            System.out.println("lastIndexOf(t, 60) = " + s.lastIndexOf('t', 60));
            System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
            System.out.println("lastIndexOf(the, 60) = " + s.lastIndexOf("the", 60));
    }
}
```

Output:

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```


Modifying a String

- Since **String** objects can not be changed, whenever we want to modify a **String**, we must either copy it into a **StringBuffer** or **StringBuilder**, or use one of the following **String** methods, which will construct a new copy of the string with our modifications complete.

substring():

- Used to extract a substring from a given string.
  - String substring(int *startIndex*)

    Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

  - String substring(int *startIndex*, int *endIndex*)

    Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

```
class StringReplace
{
    public static void main(String args[])
    {
        String org = "This is a test. This is, too.";
        String result ;


        result=org.substring(5);
        System.out.println(result);

        result=org.substring(5, 7);
        System.out.println(result);
    }
}
```

Output:

```
is a test. This is, too.
is
```


concat( ):

- This method can be used to concatenate two strings:

  String concat(String *str*)

- This method creates a new object that contains the invoking string with the contents of *str* appended to the end.

- **concat( )** performs the same function as **+**.

```
String s1 = "one";
String s2 = s1.concat("two");
```

is same as

```
String s1 = "one";
String s2 = s1 + "two";
```

replace( ):

- The first form of this method replaces all occurrences of one character in the invoking string with another character.

  String replace(char *original*, char *replacement*)

- Here, *original* specifies the character to be replaced by the character specified by *replacement*.

- For example,

  String s = "Hello".replace('l', 'w');

  puts the string “Hewwo” into **s**.

- The second form of **replace( )** replaces one character sequence with another.

  String replace(CharSequence *original*, CharSequence *replacement*)

trim( ):

- The **trim( )** method returns a copy of the invoking string from which any leading and trailing white-space has been removed.

- It has this general form:

  String trim( )

- Here is an example:

  String s = “ Hello World ".trim();

This puts the string “Hello World” into **s**.

Data Conversion Using valueOf( )

- The **valueOf( )** method converts data from its internal format into a human-readable form.

- It is a static method that is overloaded within **String** for all of Java's built-in types so that each type can be converted properly into a string.

- **valueOf( )** is also overloaded for type **Object**, so an object of any class type you create can also be used as an argument.

- Here are a few of its forms:

  static String valueOf(double *num*)
  static String valueOf(long *num*)
  static String valueOf(Object *ob*)
  static String valueOf(char *chars*[ ])

Changing the Case of Characters Within a String

- The method **toLowerCase( )** converts all the characters in a string from uppercase to lowercase.

- The **toUpperCase( )** method converts all the characters in a string from lowercase to uppercase.

- Non-alphabetical characters, such as digits, are unaffected.

- Here are the general forms of these methods:

  String toLowerCase( )
  String toUpperCase( )

Additional String Methods

Method	Description
int codePointAt(int i)	Returns the Unicode code point at the location specified by i.
int codePointBefore(int i)	Returns the Unicode code point at the location that precedes that specified by i.
int codePointCount(int start, int end)	Returns the number of code points in the portion of the invoking String that are between start and end–1.
boolean contains(CharSequence str)	Returns true if the invoking object contains the string specified by str. Returns false, otherwise.
boolean contentEquals(CharSequence str)	Returns true if the invoking string contains the same string as str. Otherwise, returns false.
boolean contentEquals(StringBuffer str)	Returns true if the invoking string contains the same string as str. Otherwise, returns false.
static String format(String fmtstr, Object ... args)	Returns a string formatted as specified by fmtstr.
static String format(Locale loc, String fmtstr, Object ... args)	Returns a string formatted as specified by fmtstr. Formatting is governed by the locale specified by loc.
boolean matches(string regExp)	Returns true if the invoking string matches the regular expression passed in regExp. Otherwise, returns false.

Method	Description
int offsetByCodePoints(int start, int num)	Returns the index with the invoking string that is num code pointsbeyond the starting index specified by start.
String replaceFirst(String regExp, String newStr)	Returns a string in which the first substring that matches the regular expression specified by regExp is replaced by newStr.
String replaceAll(String regExp, String newStr)	Returns a string in which all substrings that match the regular expression specified by regExp are replaced by newStr.
String[ ] split(String regExp)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in regExp.
String[ ] split(String regExp, int max)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in regExp. The number of pieces is specified by max. If max is negative, then the invoking string is fully decomposed. Otherwise, if max contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If max is zero, the invoking string is fully decomposed.
CharSequence subSequence(int startIndex, int stopIndex)	Returns a substring of the invoking string, beginning at startIndex and stopping at stopIndex. This method is required by the CharSequence interface, which is now implemented by String.

StringBuffer

- We know that, **String** represents fixed-length, immutable character sequences.
- In contrast, **StringBuffer** represents growable and writeable character sequences.
- We can insert characters in the middle or append at the end using this class.
- **StringBuffer** will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

Constructors:

- ***StringBuffer*** class has 4 constructors:
  - StringBuffer( ) : Reserves space for 16 characters without reallocation.
  - StringBuffer(int *size*) : accepts an integer argument that explicitly sets the size of the buffer
  - StringBuffer(String *str*) : accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation.
  - StringBuffer(CharSequence *chars*) : creates an object that contains the character sequence contained in *chars*

length( ) and capacity( ):

- These two methods can be used to find the length and total allocated capacity of ***StringBuffer*** object.

```
class StringBufferDemo
{
    public static void main(String args[])
    {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("Original string = " + sb);
        System.out.println("length = " + sb.length());        //prints 5
        System.out.println("capacity = " + sb.capacity()); //prints 21
    }
}
```


ensureCapacity( ):

- If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use this method to set the size of the buffer.

- This is useful if you know in advance that you will be appending a large number of small strings to a **StringBuffer**.

- **ensureCapacity( )** has this general form:

  void ensureCapacity(int *capacity*)

- Here, *capacity* specifies the size of the buffer.

charAt( ) and setCharAt( ):

- The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method.

- You can set the value of a character within a **StringBuffer** using **setCharAt()**.

- Their general forms are shown here:

  char charAt(int *where*)

  void setCharAt(int *where*, char *ch*)

- For **charAt( )**, *where* specifies the index of the character being obtained.

- For **setCharAt( )**, *where* specifies the index of the character being set, and *ch* specifies the new value of that character.


```
class setCharAtDemo
{
    public static void main(String args[])
    {
        StringBuffer sb = new StringBuffer("Hello");

        System.out.println("buffer before = " + sb);
        System.out.println("charAt(1) before = " + sb.charAt(1));
        sb.setCharAt(1, 'i');
        sb.setLength(2);
        System.out.println("buffer after = " + sb);
        System.out.println("charAt(1) after = " + sb.charAt(1));
    }
}
```

Output:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

getChars( ):

- To copy a substring of a **StringBuffer** into an array, use the **getChars( )** method.
- It has this general form:
  void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)
- Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring.
- This means that the substring contains the characters from *sourceStart* through *sourceEnd*–1.
- The array that will receive the characters is specified by *target*.
- The index within *target* at which the substring will be copied is passed in *targetStart*.
- Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

append( ):

- The **append( )** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object.
- It has several overloaded versions.
- Here are a few of its forms:
  StringBuffer append(String *str*)
  StringBuffer append(int *num*)
  StringBuffer append(Object *obj*)
- **String.valueOf( )** is called for each parameter to obtain its string representation.
- The result is appended to the current **StringBuffer** object.
- The buffer itself is returned by each version of **append( )** to allow subsequent calls.

```
class appendDemo
{
    public static void main(String args[])
    {
        String s;
        int a = 42;
        StringBuffer sb = new StringBuffer(40);
        s = sb.append("a = ").append(a).append("!").toString();
        System.out.println(s);
    }
}
```

Output:

42!

insert( ):

- The **insert( )** method inserts one string into another.
- It is overloaded to accept values of all the simple types, plus **String**s, **Object**s, and **CharSequence**s.
- Like **append( )**, it calls **String.valueOf( )** to obtain the string representation of the value it is called with.
- This string is then inserted into the invoking **StringBuffer** object.
- Few forms are:
  - StringBuffer insert(int *index*, String *str*)
  - StringBuffer insert(int *index*, char *ch*)
  - StringBuffer insert(int *index*, Object *obj*)
- Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object.

```
class insertDemo
{
    public static void main(String args[])
    {
        StringBuffer sb = new StringBuffer("I Java!");
        sb.insert(2, "like ");
        System.out.println(sb);
    }
}
```

Output:

I like Java!

reverse( ):

- Used to reverse the characters within a string.

```
class ReverseDemo
{
    public static void main(String args[])
    {
        StringBuffer s = new StringBuffer("abcdef");
        System.out.println(s);
        s.reverse();
        System.out.println(s);
    }
}
```

Output:

abcdef
fedcba

delete( ) and deleteCharAt( ):

- You can delete characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( )**.

- These methods are shown here:
  - StringBuffer delete(int *startIndex*, int *endIndex*)

    It deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index **one past the last character** to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*–1. The resulting **StringBuffer** object is returned.

  - StringBuffer deleteCharAt(int *loc*)

    It deletes the character at the index specified by *loc*. It returns the resulting **StringBuffer** object.


```
class deleteDemo
{
   public static void main(String args[])
   {
        StringBuffer sb = new StringBuffer("This is a test.");
        sb.delete(4, 7);
        System.out.println("After delete: " + sb);
        sb.deleteCharAt(0);
        System.out.println("After deleteCharAt: " + sb);
   }
}
```

Output:

```
After delete: This a test.
After deleteCharAt: his a test.
```

replace( ):

- You can replace one set of characters with another set inside a **StringBuffer** object by calling **replace( )**.

- Its signature is shown here:

  StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)

- The substring being replaced is specified by the indexes *startIndex* and *endIndex*.

- Thus, the substring at *startIndex* through *endIndex*–1 is replaced.

- The replacement string is passed in *str*.

- The resulting **StringBuffer** object is returned.


```
class replaceDemo
{
   public static void main(String args[])
   {
        StringBuffer sb = new StringBuffer("This is a test.");
        sb.replace(5, 7, "was");
        System.out.println("After replace: " + sb);
   }
}
```

Output:

```
After replace: This was a test.
```

substring( ) :

- You can obtain a portion of a **StringBuffer** by calling **substring( )**.

- It has the following two forms:

  String substring(int *startIndex*)

  String substring(int *startIndex*, int *endIndex*)

- The first form returns the substring that starts at *startIndex* and runs to the end of the invoking **StringBuffer** object.

- The second form returns the substring that starts at *startIndex* and runs through *endIndex*–1.

- These methods work just like those defined for **String** that were described earlier.

Additional StringBuffer Methods

Method	Description
StringBuffer appendCodePoint(int ch)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.
int codePointAt(int i)	Returns the Unicode code point at the location specified by i.
int codePointBefore(int i)	Returns the Unicode code point at the location that precedes that specified by i.
int codePointCount(int start, int end)	Returns the number of code points in the portion of the invoking String that are between start and end–1.
int indexOf(String str)	Searches the invoking StringBuffer for the first occurrence of str. Returns the index of the match, or –1 if no match is found.
int indexOf(String str, int startIndex)	Searches the invoking StringBuffer for the first occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String str)	Searches the invoking StringBuffer for the last occurrence of str. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String str, int startIndex)	Searches the invoking StringBuffer for the last occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.

Additional StringBuffer Methods

Method	Description
int offsetByCodePoints(int start, int num)	Returns the index with the invoking string that is num code points beyond the starting index specified by start.
CharSequence subSequence (int startIndex, int stopIndex)	Returns a substring of the invoking string, beginning at startIndex and stopping at stopIndex. This method is required by the CharSequence interface, which is now implemented by StringBuffer.
void trimToSize( )	Reduces the size of the character buffer for the invoking object to exactly fit the current contents.

StringBuilder

- J2SE 5 adds a new string class to Java's already powerful string handling capabilities.
- This new class is called **StringBuilder**.
- It is identical to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe.
- The advantage of **StringBuilder** is faster performance.
- However, in cases in which you are using multithreading, you must use **StringBuffer** rather than **StringBuilder**.