

Garbage Collection and useful utility classes

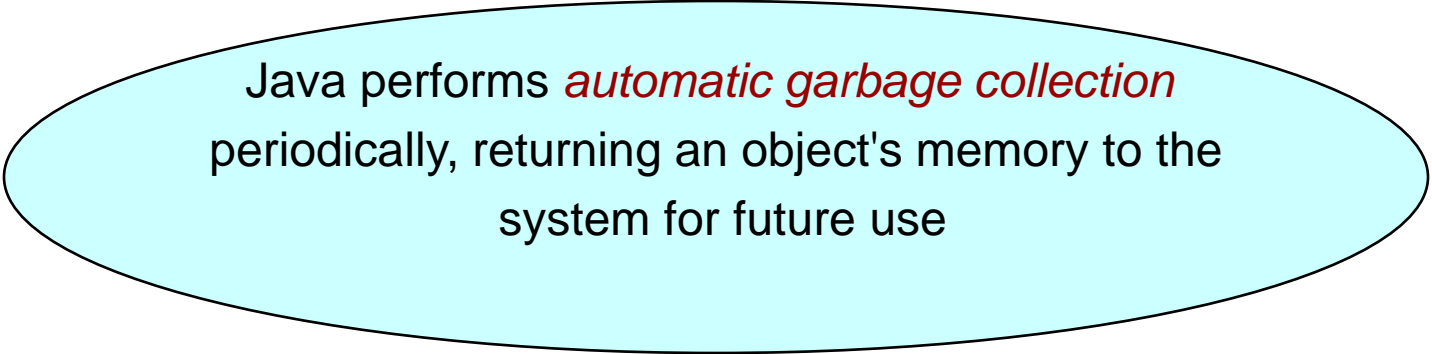
Pradeep LN

Unit 7 Objectives

- Garbage Collection
- Invisible objects vs. unreachable objects
- Finalizers
- The String class
- Splitting strings
- Internalization
- StringBuffer and StringBuilder
- Wrapper classes
- Auto boxing / unboxing
- System class
- Date and Calendar
- Understanding Enumerated Types
- Stateful and Behavioral Enumerations

Garbage Collection

- All objects occupy memory on the heap.
- Every 'new' operation results in a new memory allocation on the heap



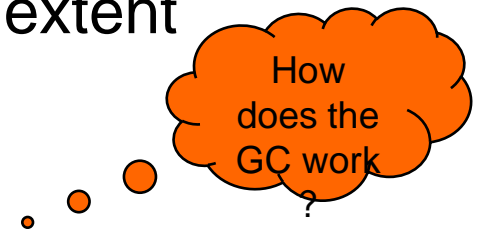
Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use

- In Java, we don't have to free the memory used by an object.
 - The garbage collector runs periodically in the background and cleans up unreferenced objects.

Automatic Garbage Collection

Automatic garbage collection has many advantages

- Eliminates the well-known problems of dangling pointers and memory leaks
 - With many efficient garbage collection algorithms that have evolved over a period of time, memory leaks can be minimized to a large extent
- Increases reliability
 - Results in code that is free of memory related bugs, which are generally very difficult to debug.



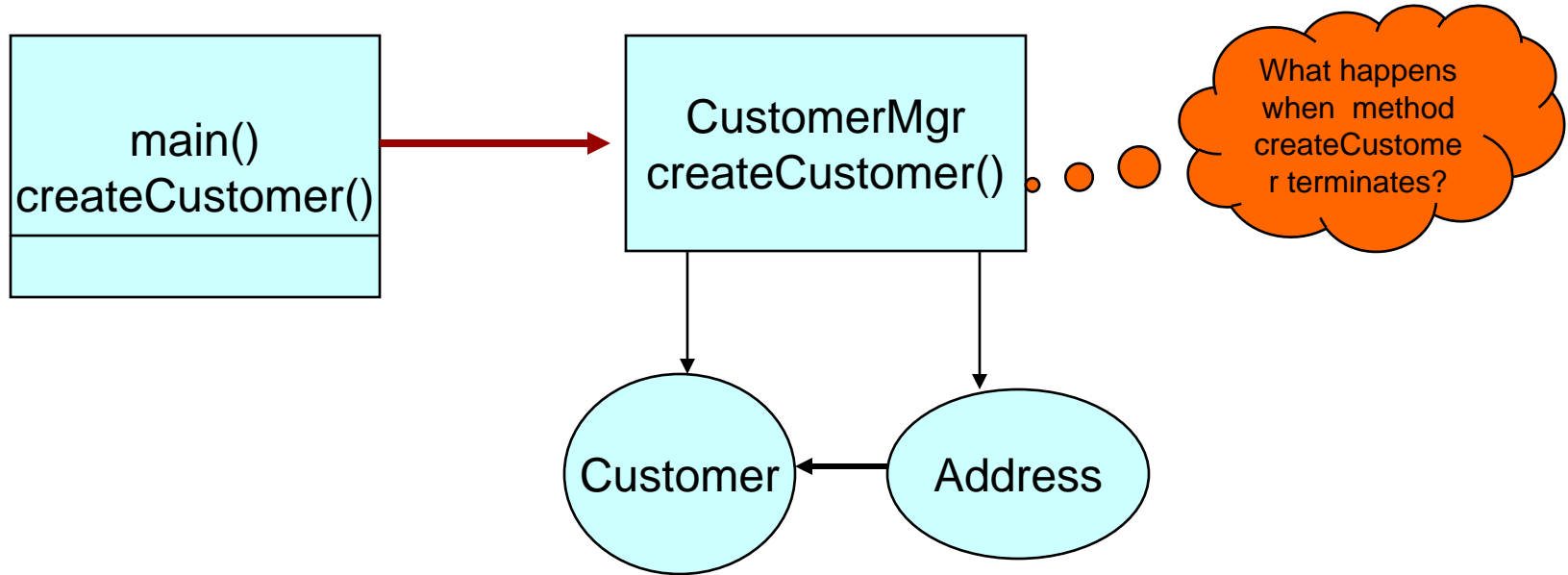
Understanding Garbage Collection

- The GC goes about reclaiming memory occupied by unreachable objects.
- An object enters an ***unreachable*** state when no more strong references to it exist.

```
class CustomerMgr
{
    public void createCustomer()
    {
        Customer customer = new Customer();
        Address custAdd = new Address();
        customer.address = custAdd;
        // do some processing
    }
}
```

```
class Client
{
    public static void main(
        String[] args)
    {
        CustomerMgr mgr =
            new CustomerMgr();
        mgr.createCustomer();
    }
}
```

Understanding Garbage Collection



- Customer and Address objects go out of scope, that is, they become unreachable to the piece of code
- These objects become eligible for garbage collection

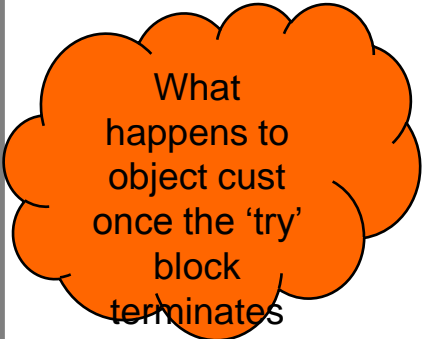
Understanding Garbage Collection

- Automatic garbage collection does not necessarily mean removing all memory management responsibility from the application developer.
 - It is most widely misunderstood this way
- Application developers are required to follow good memory management techniques like setting references to null, once they are done with working with the objects.
- Lack of this conviction may result in 'invisible objects' being created and may lead to unintentional object retention.

Invisible Objects

- Invisible objects are a result of programs 'holding on' to a reference, which is no longer required.
- Refer to the below code segment that causes an invisible object that escapes garbage collection

```
public void run()
{
    try{
        Customer cust = new Customer();
        cust.validateCustInfo();
    }
    catch (Exception e)
    {}
    while (true)
    {
        // ... loops forever
    }
}
```



What happens to object cust once the 'try' block terminates

Understanding Garbage Collection

- While object is inaccessible outside the try block, it does not still become an eligible candidate for garbage collection
 - Only when run() method terminates, does stack unwinding happen and the object becomes unreachable
- As a good programming practice, always remember to set references that are no longer required to 'null'
 - This will enable the garbage collector to reclaim memory

Invoking Garbage Collector

- Garbage collector can be invoked by invoking the method `System.gc()`
 - The request may or may not be honored
 - There is no guarantee of desired memory being reclaimed.
- The JVM specification does not lay down strict rules about how garbage collection needs to be performed
 - A rigidly defined garbage collection model might be impossible to implement on all platforms
- It is optimized differently by different vendors
- See Listing : **GCTest.java**

Performing cleanup

- Garbage collection only reclaims unused memory, while system resources used by the program need to be released by the developer.
- Every class can include the *finalize()* method, which is *invoked* (if defined) just before object destruction.
 - This presents an opportunity to perform cleanup of any special allocations made by the program.

```
protected void finalize()
{
    try {
        file.close();
        conn.close();
    }
    catch(Exception e)
    { }
}
```

The String Class

Management of Data comprising of multiple characters can be done through String Object.

String

Value
Length

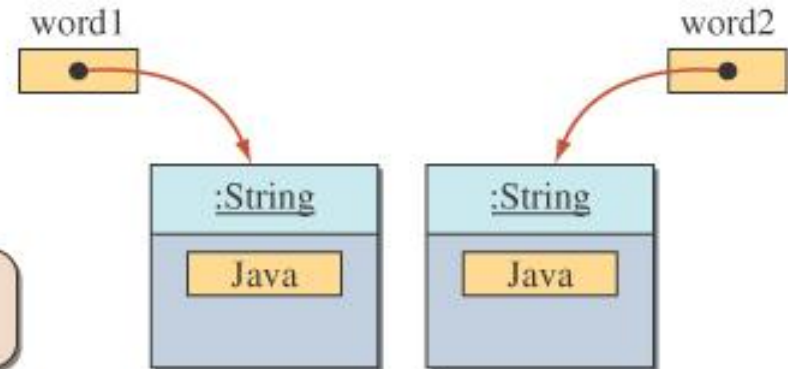
String()
String(String str);
Length():int
equals(String str): boolean
Concat(String str): String

Once a String object has been created, neither its value nor its length can be changed, **String** objects are *immutable*

Strings are immutable

```
String word1, word2;  
  
word1 = new String("Java");  
  
word2 = new String("Java");
```

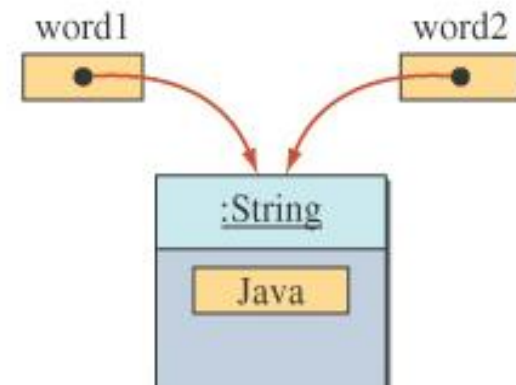
Whenever the **new** operator is used, there will be a new object.



We can do this because String objects are immutable.

```
String word1, word2;  
  
word1 = "Java";  
  
word2 = "Java";
```

Literal string constant such as "Java" will always refer to the one object.



String object

Different ways of creating Strings

```
String str;  
str = new String("hello");  
String str2 = new String(str);  
String literal = "hello";
```

String
Value ="hello" Length =5;

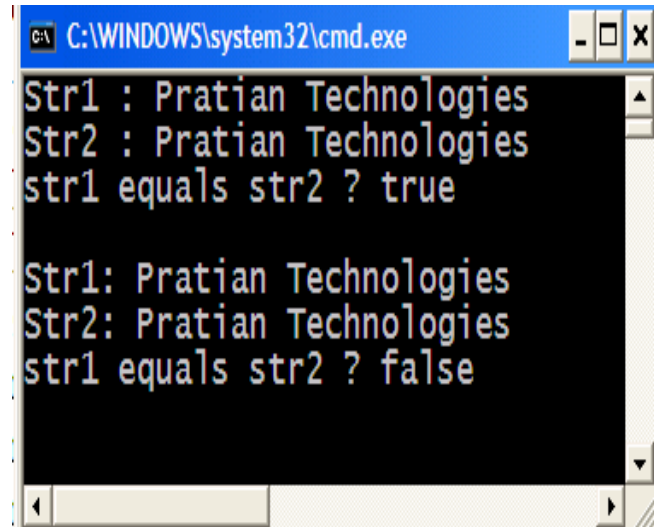
The String class has overloaded constructors to support different ways of object creation.

The class String includes methods for examining :
individual characters of the sequence, comparing strings,
searching strings, extracting substrings, creating a copy of a string

String Equality

```
class EqualsTest {  
    public static void main(String args[])  
    {  
        String str1, str2;  
        str1 = "Pratian Technologies";  
        str2 = str1;  
        System.out.println("String1 : "+str1);  
        System.out.println("String2 : "+str2);  
        System.out.print("Str1 equals Str2 ? ");  
        System.out.println(str1 == str2);  
        str2 = new String(str1);  
        System.out.println("String1: " + str1);  
        System.out.println("String2: " + str2);  
        System.out.print("str1 equals str2 ? ");  
        System.out.println(str1 == str2);  
    }  
}
```

What is
the
output ?



```
C:\WINDOWS\system32\cmd.exe  
Str1 : Pratian Technologies  
Str2 : Pratian Technologies  
str1 equals str2 ? true  
  
Str1: Pratian Technologies  
Str2: Pratian Technologies  
str1 equals str2 ? false
```

String Equality

- `==` merely checks if the two references are equal, that is, pointing to the same object in memory.
 - It does not check if the actual contents of the String objects are equal
- Comparison of String objects is done by invoking the `equals` method of String class
 - `public boolean equals(Object obj)`
 - String class overrides the `equals()` method of class Object.
 - `public boolean equalsIgnoreCase(String str)`
 - Ignores case while comparing the contents

String Concatenation

- String concatenation is done by using the '+' operator
- String class also provides a method
 - String concat(String str)
 - Concatenates the specified string to the end of this string.

```
String lastName = "Gandhi";  
String name = new String("Mohandas" + "KaramChand " + lastName);  
System.out.println("Mahatma".concat(lastName));
```

Converting to String

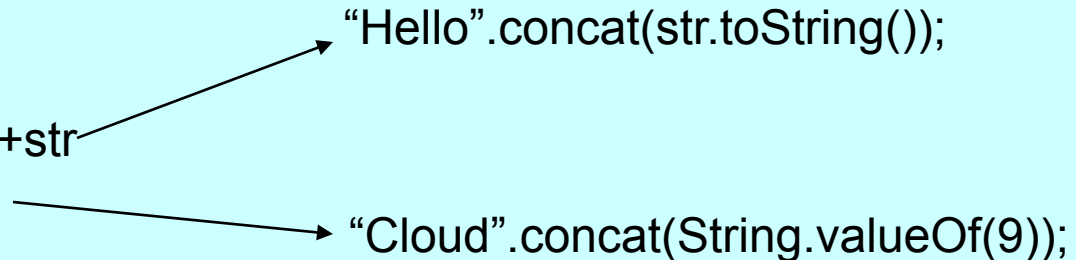
- The `String.valueOf()` method is a class method that provides the 'String' representation of primitive types and objects.
- `public static String valueOf(int integerLiteral)`
 - Overloaded methods for all primitive types
 - Overloaded method that takes an `Object` as argument

```
String number = String.valueOf(128);           // Creates "128"  
String truth = String.valueOf(true);           // Creates "true"  
String bee = String.valueOf('B');             // Creates "B"  
String pi = String.valueOf(Math.PI);          // Creates "3.14159"
```

Converting to String

- Any primitive value or object that is concatenated with a string is converted to String

```
String str = "World"  
String greeting = "Hello" + str  
String str2 = "Cloud" + 9
```



The diagram illustrates the conversion of primitive values to strings during concatenation. An arrow points from the variable `str` in the line `String greeting = "Hello" + str` to the code `"Hello".concat(str.toString());`. Another arrow points from the primitive value `9` in the line `String str2 = "Cloud" + 9` to the code `"Cloud".concat(String.valueOf(9));`.

```
"Hello".concat(str.toString());  
"Cloud".concat(String.valueOf(9));
```

Manipulating Character Case

- String class provides the following methods to manipulate character case in String.

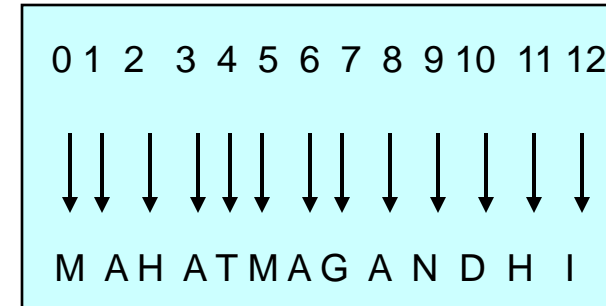
String toUpperCase()

String toLowerCase()

Original String object is returned if none of the characters, is changed, otherwise new String object is constructed and returned.

String Indexing

- Strings are indexed, starting at 0.
- String class provides methods to search for a specific character inside the string object.
 - If the search is successful, then the index of the char is returned, otherwise -1 is returned.
- `int indexOf(int c)`
 - Returns the index of first occurrence of the argument char.
- `int indexOf(int c, int fromIndex)`
 - Finds the index of the first occurrence of the argument character in a string, starting at the index specified in the second argument.
- `int indexOf(String str)`
 - Finds the start index of the first occurrence of the substring argument in a String.



Sub Strings

- String class provides methods to extract specified portion of the given String.
 - String substring(int startIndex)
 - String substring(int startIndex, int endIndex)
- A new String object containing the substring is created and returned.
 - The original String will not be affected.

Splitting Strings

- The `split()` method simplifies the task of breaking a string into substrings, or tokens.
- This method takes a delimiter separated string as argument and splits the string into several tokens based on the occurrences of the delimiter (regex)

`This;is a;delimiter;separated;string.`

`split` this string based on regex `“;”`

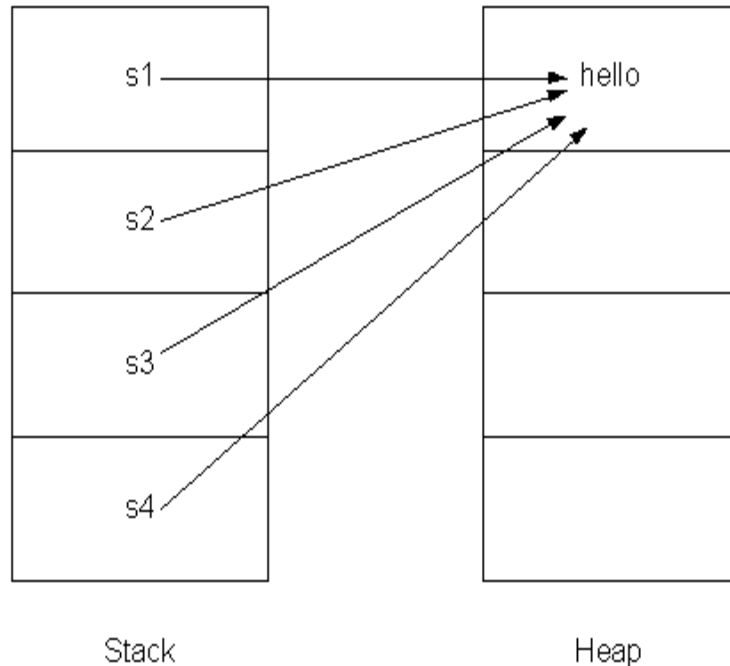
<code>strArr[0]</code>	<code>strArr[1]</code>	<code>strArr[2]</code>	<code>strArr[3]</code>	<code>strArr[4]</code>
<code>This</code>	<code>is a</code>	<code>delimiter</code>	<code>separated</code>	<code>string</code>

Splitting Strings

- `String[] split(String regex)`
 - Searches for a match as specified by the string argument (which contains a regular expression) and splits this string into an array of strings accordingly.
- `String[] split(String regex, int limit)`
 - integer argument specifies the maximum size of the returned array.

```
String expr = "Trying to split this string";  
String[] tokens = expr.split(" ");
```


String Internalization



- The JVM maintains an internal list of references (pool of unique String objects) for String literals in our application
 - To avoid duplicate String objects in heap memory.
- Whenever the JVM loads String literals from class file and executes, it checks whether that String exists in the internal list or not.
 - If it already exists in the list, then it does not create a new String and it uses reference to the existing String Object.

What about
String
objects
created on
the fly ?

String Internalization

- JVM does this type of checking internally for String literals but not for String object which it creates through 'new' keyword.
- String objects occupy a lot of memory, and to optimize memory utilization, the mechanism of string internalization is adopted.
 - This forces JVM to check the internal list and use the existing String object if it is already present.
- Interned strings avoid duplicate strings.
 - There is only one copy of each String that has been interned, no matter how many references point to it.

String Internalization

- String intern()

When the intern method is invoked, if the pool already contains a string equal to this object , then the string from the pool is returned. Otherwise, this object is added to the pool and a reference to this object is returned.

- See Listing : [StringInternTest.java](#)

- Interned Strings not only optimize memory utilization, but also ensure better performance.

- See Listing : [StringInternDemo.java](#)

String Methods

replace(char oldChr, char newChar)	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
contains(CharSequence s)	Returns true if the string contains the specified character sequence.
trim	Removes the leading and trailing spaces. str1.trim()
charAt(int index)	Returns the char value at the specified index.
startsWith	Returns true if a string starts with a specified prefix string.str1.startsWith(str2)
endsWith	Returns true if a string ends with a specified suffix string.str1.endsWith(str2)

StringBuffer

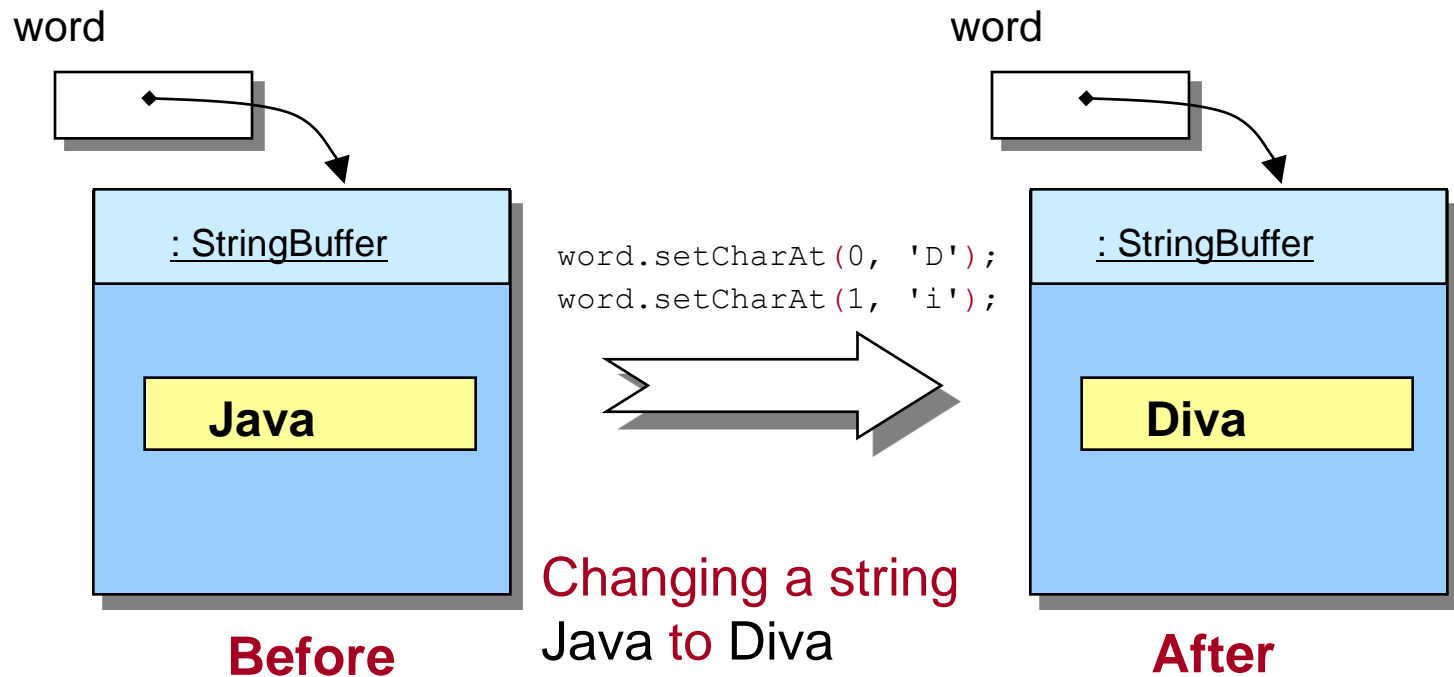
- A StringBuffer is a thread-safe, mutable sequence of characters.
- A StringBuffer is like a String, but can be modified.
 - It contains some particular sequence of characters, the length and content of the sequence can be changed through certain method calls.

In string processing applications,
we would like to change the contents of a string.
In other words, we want it to be mutable.

- See listing : **StringReverse.java**

StringBuffer

```
StringBuffer word = new StringBuffer("Java");  
word.setCharAt(0, 'D');  
word.setCharAt(1, 'i');
```



StringBuffer

Important Methods of StringBuffer

- append method
 - StringBuffer append(String s)
 - Appends the specified string to this character sequence.
 - StringBuffer append(StringBuffer sb)
 - StringBuffer append(int i)
 - Appends the string representation of the int argument to this sequence.
 - Overloaded for all primitive types

```
StringBuffer strBuf = new StringBuffer();  
strbuf.append("Hello");  
strbuf.append(" World");  
System.out.println(strbuf);
```

StringBuffer

Important Methods of StringBuffer

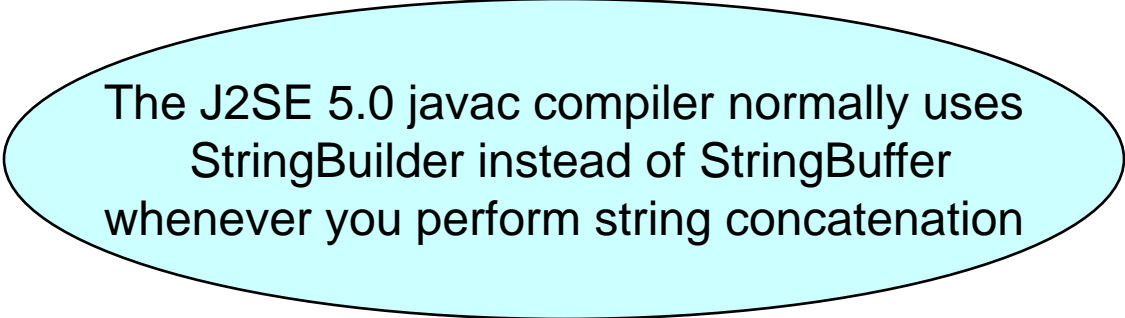
- insert method
 - StringBuffer insert(int offset, String str)
 - Inserts the string into this character sequence, at the specified position
 - StringBuffer insert(int offset, int i)
 - Inserts the string representation of the int argument into this sequence, at the specified position.

```
strbuf.insert(5, " Java ");  
System.out.println(strbuf);
```

- reverse()
 - Causes this character sequence to be replaced by the reverse of the sequence.
- String toString()
- See Listing : **StringBufferReverse.java**

StringBuilder

- Is a mutable version of String
- Introduced in Java 5.0 (SDK 1.5)
 - To improve the performance of the StringBuffer class



The J2SE 5.0 javac compiler normally uses
StringBuilder instead of StringBuffer
whenever you perform string concatenation

- StringBuilder and StringBuffer support exactly the same set of methods
- Methods of this class are not synchronized
 - No synchronization! Not safe for use by multiple threads

Wrapper classes

- Although “Everything is an object” in Java, yet we have primitive types like int, double and so on
 - For improving performance, primitive types are retained in Java
- Wrappers are classes that wrap up primitive values in classes
 - These classes offer utility methods to manipulate the values represented by the primitives
- Wrappers classes are particularly useful while working with Collections like Vector, ArrayList and others that only work with ‘Objects’

Wrapper classes

The `java.lang` package contains *wrapper classes* that correspond to each primitive type:

<u>Primitive Type</u>	<u>Wrapper Class</u>
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>char</code>	<code>Character</code>
<code>boolean</code>	<code>Boolean</code>
<code>void</code>	<code>Void</code>

Wrapper classes

- Wrapper classes offer utility methods
 - to extract the primitive value from a String
 - to convert to other primitive types
 - to represent a primitive value by its wrapper
- Some methods of class Integer
 - static int parseInt(String strNum) throws NumberFormatException
 - double doubleValue()
 - int intValue()
 - Returns the value of this Integer as an int.
- See Listing : [WrapperDemo.java](#)

Autoboxing and Unboxing

- Java 1.5 introduces to the language the feature of autoboxing / unboxing
- Wrapper types and primitive types can now be interchangeably used
- Prior to Java 1.5

Make method calls to extract primitive values from wrapper objects

```
Integer intObj = new Integer(5);  
int i = intObj.intValue();
```

explicitly wrap up primitive values into objects

- Java 1.5 onwards, the following code works

```
Integer intObj = 5;  
int i = intObj;
```

Autoboxing and Unboxing

- Autoboxing refers to the automatic storing of a value of primitive type into an object of the corresponding wrapper class.
- Unboxing refers to automatic conversion of the value represented by the wrapper object to the corresponding primitive type

```
Integer intObj = 4;  
intObj++;  
System.out.println(intObj);  
int i = intObj;
```

Compiler generates

```
Integer intObj = new  
                    Integer(4);  
intObj = new Integer(  
    (intObj.intValue())++);  
System.out.println  
    (intObj.intValue());  
int i = intObj.intValue();
```

System class

- The System class contains several useful class fields and methods.
- System class cannot be instantiated.
 - Has no public constructor
- System class provides
 - Standard input stream
 - Standard output stream
 - Standard error stream
 - access to externally defined properties / environment variables
 - method to terminate the application

System class

- Important class fields
 - `System.out`
 - `PrintStream` to write bytes of data to the standard output device, usually the monitor
 - `System.in`
 - `InputStream` to read bytes of data from the standard input device, usually the keyboard
 - `System.err`
 - `PrintStream` to write bytes of data to the standard error device, by default is the monitor

System class

- Important Methods
 - static long currentTimeMillis()
 - Returns the current time in milli seconds
 - static void exit(int status)
 - Terminates the currently running Java Virtual Machine.
 - static void gc()
 - Runs the garbage collector
 - static void runFinalization()
 - Runs the finalization methods of any objects pending finalization.

System Properties

- System class maintains a Properties object that describes the configuration of the current working environment.
- System properties are key/value pairs maintained by the System class.
 - Define traits or attributes of the current working environment.
- When the Java virtual machine first starts up, the system properties are initialized to contain information about the runtime environment
- System class stores information like
 - the current user
 - the current version of the Java runtime
 - the character used to separate components of a file

System Properties

KEY	VALUE
"file.separator"	Character that separates components of a file path. This is "/" on UNIX and "\" on Windows.
"java.class.path"	Path used to find directories and JAR archives containing class files.
"java.home"	Installation directory for Java Runtime Environment (JRE)
"java.version"	JRE version number
"os.name"	Operating system name
"user.dir"	User working directory
"user.home"	User home directory

Reading System Properties

- The System class has two methods used to read system properties
 - static String getProperty(String key)
 - Gets the system property indicated by the specified key.
 - static Properties getProperties()
 - Returns all the current system properties.

```
String pathSeperator = System.getProperty("path.separator");  
String version = System.getProperty("java.version");  
String country = System.getProperty("user.country");  
  
Properties p = System.getProperties();
```

Setting System Properties

- System properties can be programmatically modified by invoking the method
 - static String setProperty(String key, String value)
 - Sets the system property indicated by the specified key to the specified value

```
System.setProperty("user.dir", "D:\\JavaProj\\src");  
System.setProperty("user.name", "Sindhu");
```

- System properties can also be set while launching the JVM, with use of the -D option
java -Dlang=en MyApp
- See Listing : **SystemPropertiesDemo.java**

Date & Calendar

- The Date and Calendar classes represents a specific instance in time
- These classes are in java.util package
- Methods of Date class have been deprecated and replaced by methods in Calendar class
- Calendar is an abstract class, whose getInstance() method can be used to get a calendar instance that represents current date and time
- See Listing : **DateCalendarDemo.java**

Enumerated Types

- Java SE 5.0 allows developers to declare a type as an 'enum' type.
- An enum type is a type whose fields consist of a fixed set of constants.
- For instance, days of the week can take one of the seven legal values
- This can be modeled as an enum as below
 - **enum** WEEK { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }

Prior to Java 1.5

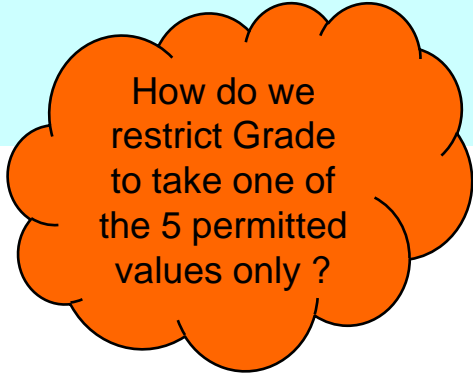
- Let us suppose that a student can score one of the five grades A, B, C, D and E
- A frequent pattern in Java was the use of set of static final variables

```
interface Grade
{
    public static final GRADE_A = 'A';
    public static final GRADE_B = 'B';
    public static final GRADE_C = 'C';
    public static final GRADE_D = 'D';
    public static final GRADE_E = 'E';
}
```


Why Enums ?

```
class Student
{
    private int StudId;
    private String name;

    public void setGrade(char ch)
    {
        grade = ch;
    }
    // Other Methods
}
```



How do we
restrict Grade
to take one of
the 5 permitted
values only ?

```
class Client
{
    public static void main(String[] args)
    {
        Student s = new Student();
        s.setStudName("Alex");
        s.setGrade(Grade.GRADE_A);
    }
}
```

//Will the below line of code execute ?
s.setGrade('Z');

Enumerated Types

- Enum helps prevent accidentally using an illegal value where a group of predefined constant values, and nothing else, are expected

```
class Student
{
    private int StudId;
    private String name;
    private Grade grade;

    public void setGrade(Grade g)
    {
        grade = g;
    }
    // Other Methods
}
```

```
enum Grade { A, B, C,
D, E; }
```

```
class Client
{
    public static void main(String[] args)
    {
        Student s = new Student();
        s.setStudName("Alex");
        s.setGrade(Grade.A);
    }
}
```

Enumerated Types

- Enums provide compile-time type safety
- As enums represent constants, the names of an enum type's fields are in uppercase letters (By convention)
- Enums can be used in switch-case structure
- See Listing : [EnumDemo.java](#)

Iterating Over Enums

- The enumerated types, has built-in method values().
- This method provides access to all of the types within an enum.

```
public void listGradeValues()  
{  
    Grade[ ] gradeValues = Grade.values( );  
    for (Grade g : Grade.values( ))  
    {  
        System.out.println("Allowed value: " + g + "");  
    }  
}
```

Structural & Behavioral Enums

- The enum declaration defines a class
 - All enums implicitly extend `java.lang.Enum`.
- The enum class body can include methods and other fields.
 - The compiler adds some methods including `values()` methods by default
- The enum class body can include constructors as well
- See Listing : [Planet.java](#)

Question time

