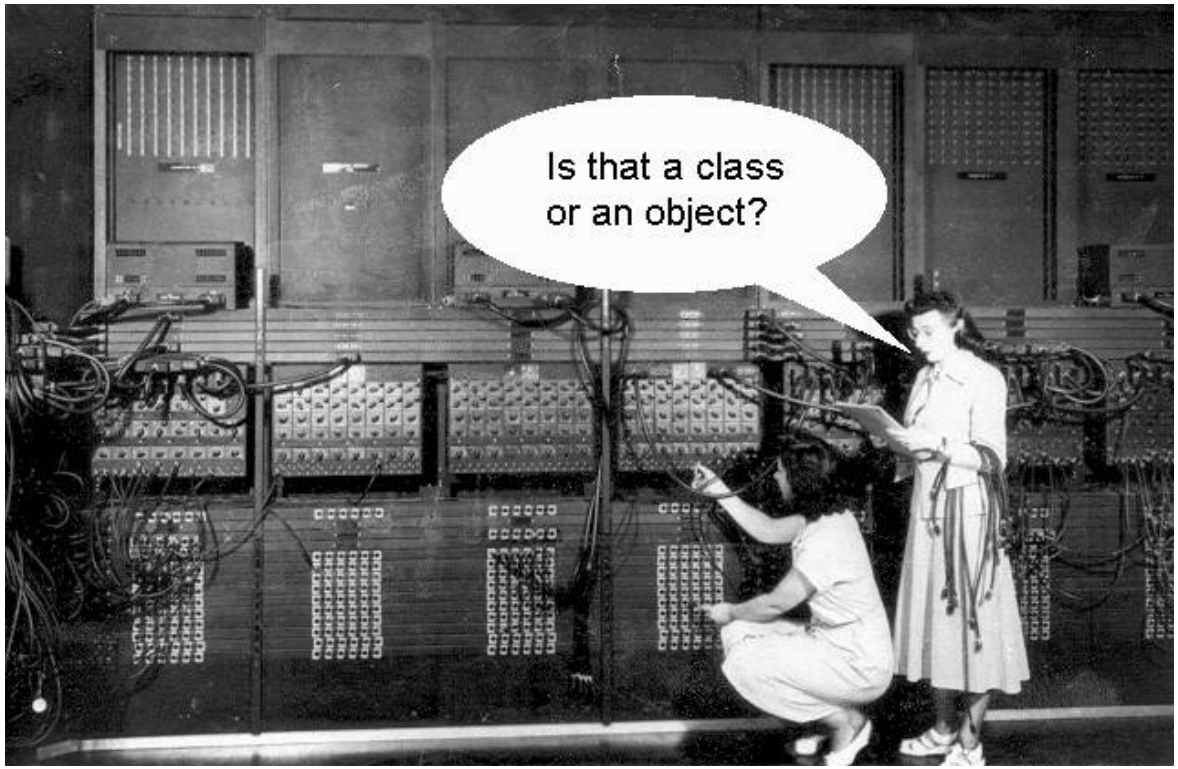
Java Strings



What is a String?

- A string is a sequence of characters treated as a unit
  - Remember that all characters here are in unicode (16 bits/char)
- Strings in Java
  - standard objects with built-in language support
    - **`String`** - class for immutable (read-only) strings
    - **`StringBuffer`** - class for mutable strings
  - can be converted to other types like integers and booleans
  - like other classes, `String` has constructors and methods
  - unlike other classes, `String` has two operators, `+` and `+=` (used for concatenation)
  - Strings, once created, cannot be modified!!
    - However, you can carry out operations on a string and save it as another string

Basic `String` Operations

- Constructors
  - **`public String()`**
    - construct a new String with the value " "
  - **`public String(String value)`**
    - construct a new String that is a copy of the specified **`String`** object value
- Basic methods
  - **`Length()`** - return the length of string
  - **`charAt()`** - return the char at the specified position
  - e.g.,

```
for(int i = 0; i < str.length(); i++)
        counts[str.charAt(i)]++;
```

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

Basic `String` Operations

- Basic methods (cont'd)
  - **`indexOf()`** - find the first occurrence of a particular character or substring in a string
  - **`lastIndexOf()`** - find the last occurrence of a particular character or substring in a string
  - e.g.,

```
static int countBetween(String str, char ch) {
      int begPos = str.indexOf(ch);
      if(begPos < 0)    // not there
            return -1;
      int endPos = str.lastIndexOf(ch);
      return (endPos - begPos -1);
}
```

Return the number of characters between the first and last occurrences of *ch*

Basic `String` Operations

- Overloaded **`indexOf`** and **`lastIndexOf`** methods

Method	Returns index of...
`indexOf(char ch)`	first position of ch
`indexOf(char ch, int start)`	first position of ch > start
`indexOf(String str)`	first position of str
`indexOf(String str, int start)`	first position of str > start
`lastIndexOf(char ch)`	last position of ch
`lastIndexOf(char ch, int start)`	last position of ch < start
`lastIndexOf(String str)`	last position of str
`lastIndexOf(String str, int start)`	last position of str < start

Literal Strings

- are anonymous objects of the String class
- are defined by enclosing text in double quotes: “This is a literal String”
- don’t have to be constructed
- can be assigned to String variables
- can be passed to methods and constructors as parameters
- have methods you can call

```
//assign a literal to a String variable
String name = “Robert”;

//calling a method on a literal String
char firstInitial = “Robert”.charAt(0);

//calling a method on a String variable
char firstInitial = name.charAt(0);
```

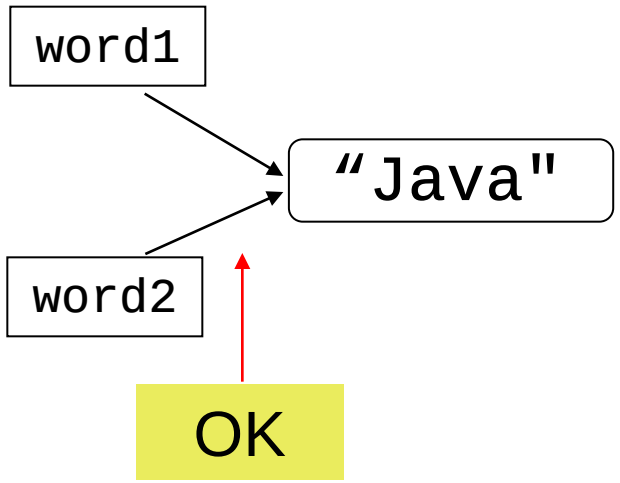
String Immutability

- Once created, a string cannot be changed: none of its methods changes the string
- Such objects are called immutable
- Immutable objects are convenient because several references can point to the same object safely: there is no danger of changing an object through one reference without the others being aware of the change

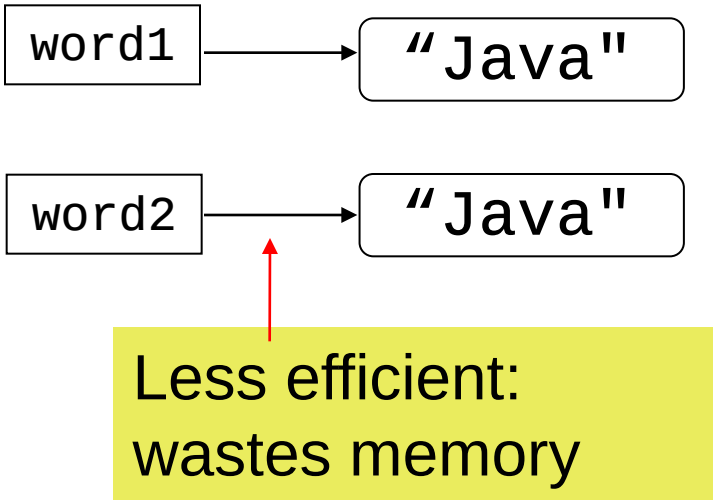
Advantages Of Immutability

Uses less memory

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



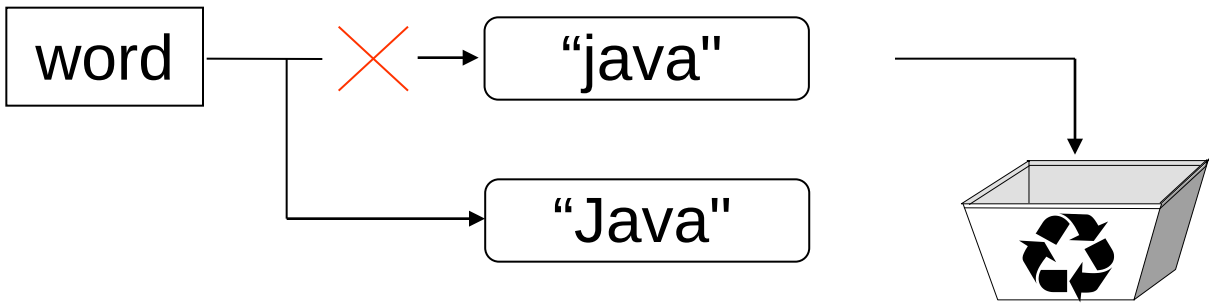

```
String word1 = “Java";
String word2 = new String(word1);
```



Disadvantages of Immutability

Less efficient — you need to create a new string and throw away the old one even for small changes

```
String word = “Java";
char ch = Character.toUpperCase(word.charAt (0));
word =  ch + word.substring (1);
```



Empty Strings

- An empty `String` has no characters. It's length is 0.

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

Empty strings

- Not the same as an uninitialized `String`

```
private String errorMsg;
```

errorMsg is **null**

- No-argument constructor creates an empty `String` (Rarely used)

```
String empty = new String();
```

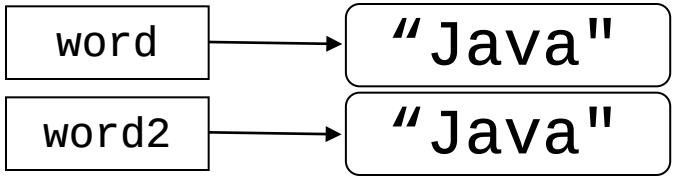
- A more common approach is to reassign the variable to an empty literal String (Often done to reinitialize a variable used to store input)

```
String empty = “”; //nothing between quotes
```

Copy Constructors

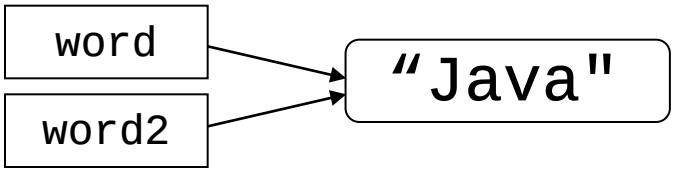
- Copy constructor creates a copy of an existing `String` (Also rarely used)
  - Not the same as an assignment
- Copy Constructor: Each variable points to a different copy of the `String`

```
String word = new String("Java");
String word2 = new String(word);
```



- Assignment: Both variables point to the same `String`

```
String word = "Java";
String word2 = word;
```

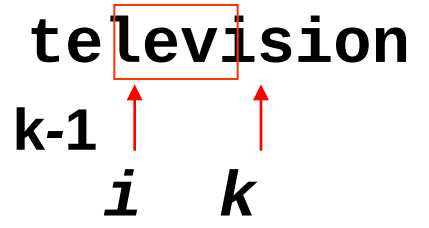


- Most other constructors take an array as a parameter to create a `String`

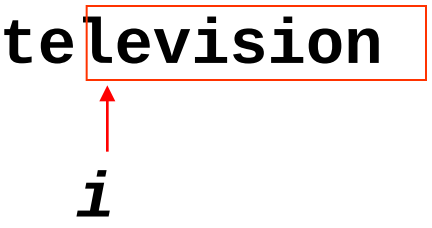
```
char[] letters = {'J', 'a', 'v', 'a'};
String word = new String(letters);//"Java"
```

Substrings

- Returns a new String by copying characters from an existing `String`
- `String subs = word.substring (i, k);`
  - returns the substring of chars in positions from **i** to **k-1**



- `String subs = word.substring (i);`
  - returns the substring from the **i**-th char to the end




```
"television".substring (2,5);
"immutable".substring (2);
"bob".substring (9);
```

Returns:

```
"lev"
"mutable"
"" (empty string)
```

Substrings Example

- E.g., extracting quoted substrings from another string

```
public static String quotedString(String from, char start,
                                  char end)
{
    int startPos = from.indexOf(start);
    int endPos = from.lastIndexOf(end);
    if(startPos == -1)  return null; // no start found
    else if(endPos == -1)            // no end found
        return from.substring(startPos);
    else                             // both start and end found
        return from.substring(startPos, endPos + 1);
}
```

quotedString(“Say <Bonjour!> to the class”, ‘<’, ‘>’);
➔ <Bonjour!>

String Concatenation

```
String word1 = “re”, word2 = “think”; word3 = “ing”;
int num = 2;
String result = word1 + word2;
   //concatenates word1 and word2   “rethink“
String result = word1.concat (word2);
   //the same as word1 + word2  “rethink“
result += word3;
   //concatenates word3 to result  “rethinking”
result += num; //converts num to String
  //and concatenates it to result  “rethinking2”
```

String Comparisons

- Methods for entire strings
  - **`equals()`** - return true if two strings have the same length and exactly the same Unicode characters
  - **`equalsIgnoreCase()`** - compare strings while ignoring case
  - **`compareTo()`** - create an internal canonical ordering of strings
- e,g.,
  - `boolean b = word1.equals(word2);`

    returns true if the string word1 is equal to word2
  - `boolean b = word1.equalsIgnoreCase(word2);`

    returns true if the string word1 matches word2, case-blind

```
b = “Raiders”.equals(“Raiders”);//true
b = “Raiders”.equals(“raiders”);//false
b = “Raiders”.equalsIgnoreCase(“raiders”);//true
```

```
if(team.equalsIgnoreCase(“raiders”))
      System.out.println(“Go You “ + team);
```

String Comparisons

- `int diff = word1.compareTo(word2);`
  returns the "difference" **word1 - word2**
- `int diff = word1.compareToIgnoreCase(word2);`
  returns the "difference" **word1 - word2**, case-blind

Usually programmers don't care what the numerical "difference" of **word1 - word2** is, just whether the difference is negative (word1 comes before word2), zero (word1 and word2 are equal) or positive (word1 comes after word2). Often used in conditional statements

```
if(word1.compareTo(word2) > 0){
      //word1 comes after word2…
}
```

String Comparison Examples

```
//negative differences
diff = “apple”.compareTo(“berry”);//a before b
diff = “Zebra”.compareTo(“apple”);//Z before a
diff = “dig”.compareTo(“dug”);//i before u
diff = “dig”.compareTo(“digs”);//dig is shorter
```

```
//zero differences
diff = “apple”.compareTo(“apple”);//equal
diff = “dig”.compareToIgnoreCase(“DIG”);//equal
```

```
//positive differences
diff = “berry”.compareTo(“apple”);//b after a
diff = “apple”.compareTo(“Apple”);//a after A
diff = “BIT”.compareTo(“BIG”);//T after G
diff = “huge”.compareTo(“hug”);//huge is longer
```

String Comparison Example: Binary search

- Binary search lookup method for a class that has a sorted array of strings

```
private String[] table;
public int position(String key) {
      int lo = 0, hi = table.length - 1;
      while(lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            int cmp = key.compareTo(table[mid]);
            if(cmp == 0)
                  return mid;    // found it!
            else if(cmp < 0)
                  hi = mid - 1 ;  // search the lower part
            else
                  lo = mid + 1;  // search the upper part
    }
    return -1;           // not found
}
```

More String Comparisons

- Methods for matching regions of strings
  - **`public boolean regionMatches(int start, String other, int ostart, int len)`**
    - return true if the given region of this String matches the given region of the string other.
  - **`public boolean regionMatches(boolean ignoreCase,int start, String other, int ostart, int len)`**
- Other methods
  - **`public boolean startsWith(String prefix,int toffset)`**
    - return true if this String starts (at toffset) with the given prefix
  - **`public boolean startsWith(String prefix)`**
    - this is a shorthand for startsWith(prefix, 0)
  - **`public boolean endsWith(String prefix)`**
    - return true if this String ends with the given suffix

Utility Methods

- Two special method
  - **`hashCode()`** - return a hash based on the contents of the string
  - **`intern()`**
    - return a `String` that has the same contents as the one it is invoked on
    - compare `String` *references* to test equality, instead of the slower test of string *contents*

E.g., intern

```
int putIn(String key) {
    String unique = key.intern();
    int i;
    for(int i=0; i < tableSize; i++)
    // see if it's in the table already
        if(table[i] == unique)    return i;
    table[i] = unique;
    // it's not there -- add it in
    tableSize++;
    return i;
}
```

Making Related Strings

- The rest of the “related string” methods
  - **`public String replace(char oldChar, char newChar)`**
    - return a new `String` formed from original by replacing all occurrences of `oldCh` with `newCh`
  - **`public String trim()`**
    - return a new `String` with any leading and trailing white space removed
  - **`public String toLowerCase()`**
  - **`public String toUpperCase()`**
    - returns a new `String` formed from original by converting its characters to upper (lower) case

String Replacement Examples

```
String word1 = " Hi Bob ";
String word2 = word1.trim();
//word2 is "Hi Bob" – no spaces on either end
//word1 is still " Hi Bob " – with spaces
```

```
String word1 = "rare";
String word2 = "rare".replace('r', 'd');
//word2 is "dade", but word1 is still "rare"
```

```
String word1 = "HeLLo";
String word2 = word1.toUpperCase();//"HELLO"
String word3 = word1.toLowerCase();//"hello"
//word1 is still "HeLLo"
```

Replacements: Common Bugs

- Example: to “convert” word1 to upper case, replace the reference with a new reference

```
word1 = word1.toUpperCase();
```

- A common bug:

```
word1.toUpperCase();
```

word1 remains unchanged

`String` Conversions

- The type converted to has the method that does the conversion
  - e.g., converting from a **`String`** to an `integer` requires a static method in class `Integer`

Type	To String	From String
boolean	String.valueOf(boolean)	new Boolean(String).booleanValue()
int	String.valueOf(int)	Integer.parseInt(String, int base)
long	String.valueOf(long)	Long.parseLong(String, int base)
float	String.valueOf(float)	new Float(String).floatValue()
double	String.valueOf(double)	new Double(String).doubleValue()

<type conversion table>

- Type conversion of non built-in class
  - support encoding and decoding by having a **`toString`** method, and a constructor creating a new object given the string description
  - classes with a **`toString`** method can be used with ***`valueOf`***

Example: Converting Numbers to Strings

- Three ways to convert a number into a string:
  - `String s = "" + num;`

```
s = “” + 123;//”123”
```

  - `String s = Integer.toString (i);`
  - `String s = Double.toString (d);`

Integer and **Double** are “wrapper” classes from **java.lang** that represent numbers as objects. They also provide useful static methods.

```
s = Integer.toString(123);//”123”
s = Double.toString(3.14); //”3.14”
```

  - `String s = String.valueOf (num);`

```
s = String.valueOf(123);//”123”
```

The `Character` Class

- The **`Character`** class is handy for determining what type a character is (eg. letter, digit etc.)
- Most **`Character`** class methods are static allowing you to test characters without creating a new class
- Some methods include
  - **`Character.isDefined(c);`**
    - true if character is in the Unicode character set
  - **`Character.isDigit(c);`**
  - **`Character.isJavaIdentifierStart(c);`**
    - determines whether the character can be used as the first letter of a Java identifier: a letter, underscore (_), or a dollar sign ($)

The Character Class

- Some methods include (cont.)
  - `Character.isJavaIdentifier(c); // a digit or isJavaIdentifierStart()`
  - `Character.isLetter(c)`
  - `Character.isLetterOrDigit(c)`
  - `Character.isLowerCase(c)`
  - `Character.isUpperCase(c)`
  - `char Character.toLowerCase(c)`
  - `char Character.toUpperCase(c)`

Strings and `char` Arrays

- A **`String`** maps to an array of **`char`**, and vice versa
  - e.g., squeeze out all occurrences of a character from a string

Convert a `String` to an array of *`char`*

```
public static String squeezeOut(String from, char toss) {
    char[] chars = from.toCharArray();
    int len = chars.length;
    for(int i = 0; i < len; i++)
        if(chars[i] == toss) {
            --len;
            System.arrarcopy(chars, i+1, chars, len-i);
            --i;  // reexamine this spot
        }
    return new String(chars, 0, len);
}
```

Strings and `char` Arrays

- Other methods
  - **`public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`**
    - copy characters from this **`String`** into the specified array
    - any access outside the bounds of either **`String`** or **`char`** array throws an *`IndexOutOfBoundsException`*

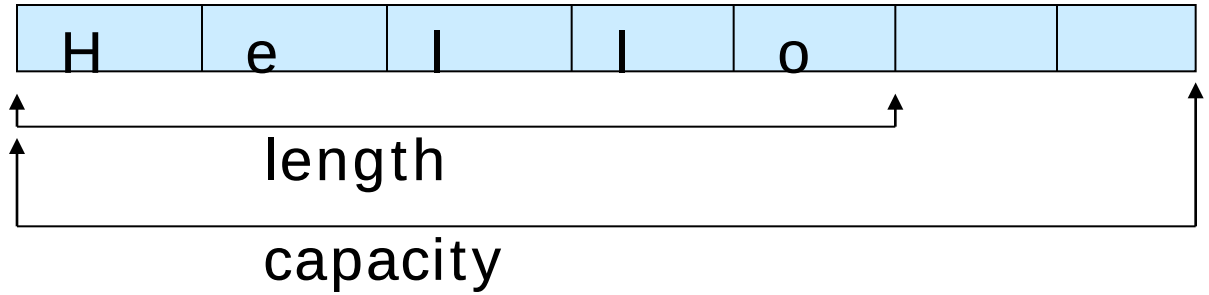
Strings and `byte` Arrays

- There are methods to convert arrays of 8-bit characters to and from 16-bit Unicode **`String`** objects
- Related methods
  - **`public String(byte[] bytes, int hiByte, int offset, int count)`**
    - this constructor makes a new **`String`** whose value is the specified subarray of the array **`byte`**s, starting at *offset* and having *count* characters
  - **`public String(byte[] bytes, int hibyte)`**
    - this is a shorthand for *`String(bytes, hibyte, 0, bytes.length)`*
  - **`public void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)`**
    - copy part of this string into array *`dst`*, starting at `dst[dstBegin]`
    - the top 8 bits of each character in the string are lost

The `StringBuffer` Class

- **`StringBuffer`** class
  - useful in building and modifying a string
  - **`StringBuffer`** does not extend **`String`** or vice versa
    - they are independent classes
  - constructors
    - **`public StringBuffer()`** - construct a **`StringBuffer`** with an initial value of “”
    - **`public StringBuffer(String str)`**
  - **`StringBuffers`** have both a **`capacity`** and a **`length`**
  - The **`capacity`** indicates the size of the **`char`** array being used to store the string
  - The **`length`** of the **`StringBuffer`** indicates the length of the string stored within the **`char`** array
  - The **`capacity`** must be at least the same size as the **`length`** or larger

Creating a `StringBuffer` Object



- The **`StringBuffer`** class provides three constructors
  - Create a **`StringBuffer`** with no characters in it (zero **`length`**) and an initial **`capacity`** of 16 characters

```
buf1 = new StringBuffer();
```

  - Create a **`StringBuffer`** with no characters in it and an initial **`capacity`** of 10 characters

```
buf2 = new StringBuffer(10);
```

  - Create a **`StringBuffer`** with the characters contained within the **`String`** passed to the constructor. The initial **`capacity`** is the length of the **`String`** + 16

```
buf3 = new StringBuffer("hello");
```

Creating a `StringBuffer` Object

- To access the length and capacity of the **`StringBuffer`** the **`StringBuffer`** class provides the methods **`length()`** and **`capacity()`**
- To ensure a minimum capacity of the **`StringBuffer`** the **`ensureCapacity()`** method can be called
  - **`s.ensureCapacity(30);`**
- The length of the string can be set using the **`setLength()`** method
  - **`s.setLength(10);`**
- If the specified length is less than the current length of the string then the string is truncated
- If the specified length is greater than the current length of the string then null characters are added to the end of the string

Using a `StringBuffer` Object

- The **`StringBuffer`** provides the methods **`charAt()`** and **`getChars()`** which perform the same as the **`String`** methods
- **`StringBuffer`** also provides the method **`setCharAt()`** to set a character at a specified location
  - **`void setCharAt(int index)`**
- The contents of the **StringBuffer** can be reversed using the **reverse()** method
  - **`void reverse()`**
- **`StringBuffer`** provides an **`append()`** method to add a string to the end of the string
- The **`StringBuffer`** class provides 10 overloaded **`append`** methods which allow various data type values to be appended to the end of a **`StringBuffer`**
  - **`void append(int i)`**
  - **`void append(char c)`**
  - **`void append(String s)`**
  - **`void append(Double d)`**

Using a `StringBuffer` Object

- In fact Java uses the **`StringBuffer`** class to append when the **+** operator is used
  - **`String s = “BC” + 22;`**
- becomes
  - **`String s = new StringBuffer(“BC”).append(22).toString();`**
- Java also uses it for the **+=** operator
  - **`s += “!”;`**
- becomes
  - **`s = new StringBuffer(s).append(“!”).toString();`**
- **StringBuffer** also provides the **insert** method to insert into the string
- Like **append**, **insert** comes in 10 varieties for different data types
  - **`void insert(int index, int i)`**
  - **`void insert(int index, char c)`**
  - **`void insert(int index, String s)`**
  - **`void insert(int index, Double d)`**

Modifying the Buffer

- Modification of the buffer of a **`StringBuffer`** object
  - e.g., **`replace`** method that does what **`String.replace`** does, except that it uses a **`StringBuffer`** object

```
public static void replace(StringBuffer str, char from,
                                              char to)
{
    for(int i = 0; i < str.length(); i++)
        if(str.charAt(i) == from) str.setCharAt(i, to);
}
```

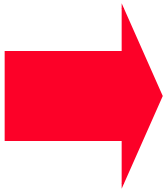
`replace` method need not to create a new object to hold the results, so successive `replace` calls can operate on one buffer

Modifying the Buffer

- **`append`** and **`insert`** methods
  - return the **`StringBuffer`** object itself
  - e.g., describe the square root of an integer

```
String sqrtInt(int I) {
    StringBuffer buf = new StringBuffer();

    buf.append("sqrt(").append(I).append(')');
    buf.append(" = ").append(Math.sqrt(I));
    return buf.toString();
}
```



`sqrt(9) = 3`

E.g., insert

- The following method is to put the current date at the beginning of a buffer

```
public static StringBuffer addDate(StringBuffer buf) {
    String now = new java.util.Date().toString();
    buf.ensureCapacity(buf.length() + now.length() + 2);
    buf.insert(0, now).insert(now.length(), " : ");
    return buf;
}
```

`insert` methods take two parameter,
namely the index to insert at and the value to insert

Modifying the Buffer

- `reverse` method
  - reverse the order of characters in the **`StringBuffer`**
  - e.g, **`reverse(“good”) = doog`**

Getting Data Out

- To get a **`String`** object from a **`StringBuffer`** object, invoke **`toString`**
- Other methods
  - **`public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`**
    - analogous to **`String.getChars`**
    - copy the characters of the specified part of the buffer (determined by `srcBegin` and `srcEnd`) int the array `dst`, starting at *`dst[dstBegin]`*

E.g., getChars

- The following method uses **`getChars`** to remove part of a buffer

```
public static StringBuffer remove(StringBuffer buf, int pos,
                                                    int cnt)
{     if(pos < 0 || cnt < 0 || pos + cnt > buf.length())
            throw new IndexOutOfBoundsException();
      int leftover = buf.length() - (pos + cnt);
      if(leftover == 0) {    // a simple truncation
            buf.setLength(pos);
            return buf; }
      char[] chrs = new char[leftover];
      buf.getChars(pos + cnt, buf.length(), chrs, 0);
      buf.setLength(pos);
      buf.append(chrs);
      return buf;
}
```

```
remove("Ping Pong", 5, 3) = Pingng
```

Capacity Management

- Capacity management
  - the buffer of a **`StringBuffer`** object grows automatically as characters are added
  - but, efficient to specify the size of the buffer

- Related methods
  - **`public StringBuffer(int capacity)`**
  - **`public synchronized void ensureCapacity(int minimum)`**
  - **`public int capacity()`**

E.g., Capacity Management

- The following method is a rewrite of the **`sqrtInt`** to allocate new space for the buffer at most once

```
String sqrtIntFaster(int i) {
    StringBuffer buf = new StringBuffer(50);
    buf.append("sqrt(").append(i).append(')');
    buf.append(" = ").append(Math.sqrt(i));

    return buf.toString();
}
```

The only change is to use a constructor that creates a `StringBuffer` object large enough to contain the result string

StringBuffer vs String

- The **`StringBuffer`** class is used when the string represented by **`StringBuffer`** needs to be modified
- String objects are constant strings and **`StringBuffer`** objects are modifiable strings
- Java distinguishes between constant strings (**`String`**) and modifiable strings (**`StringBuffer`**) for optimisation purposes

Problem

- Consider a string of words with many blanks between words. We have to write a java program which will display the string with only one space between words
  - For simplicity, we assume that there is no leading or trailing blanks

- Example :
  - If the string is "get rid of extra blanks"
  - We want to get "get rid of extra blanks"

Recursive Solution

```
String s1 =
public String function(String my_string)
{
      boolean flag = true;
      if(my_string.indexOf("  ") != -1)
           return function(my_string.substring(0,
                                  my_string.indexOf("  ") + 1)
            + my_string.substring(my_string.indexOf("  ")
            + 2));
       else return my_string;
}
```

"get rid of" → "get " " rid of"

Recursive Solution

```
public void paint( Graphics g )
{
      g.drawString("The original string is  : " + s1 ,
            25, 75 );
      g.drawString("The function returns :" +
            function( s1 ), 25, 125 );

}
```

Second Solution

0. Start with an empty string
   - This will ultimately hold the result
   - We will call this `result_string`

1. While there are more words to process

2. Pick next word

3. Append it to the result string with a single space, if needed

4. Return the result as a string

Observations

- A string object can't change!! This can be handled in two ways :
  1. Create a `result_string` to hold the current sentence
     - To append a new word, create a new string containing `result_string` concatenated to the new word and call it `result_string`
  2. Use `StringBuffer`
     - `StringBuffers` allow the string to be modified

StringBuffer methods

- What happens if you say :

```
buf = new StringBuffer("Computer Science");
buf.append(' X');
buf.append("Windsor");
buf.insert(3, " Subir ");
```

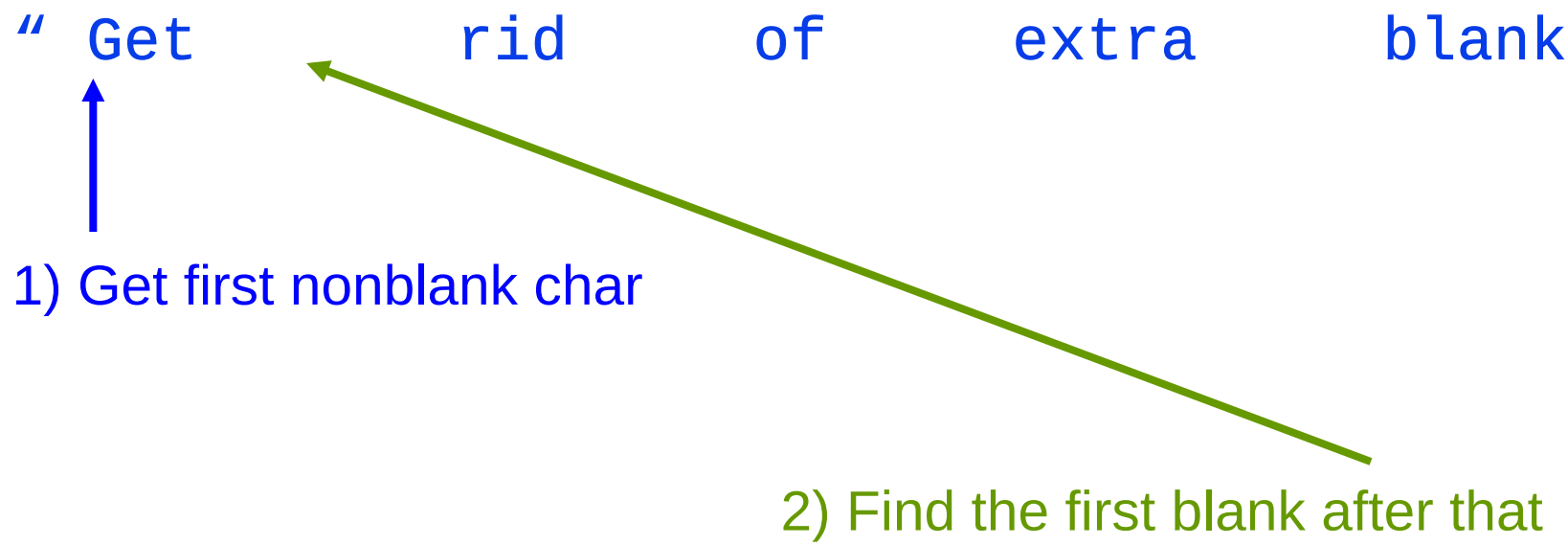
"Computer Science" → "Computer ScienceX"

→ "Computer ScienceXWindsor" →

"Com Subir puter ScienceXWindsor"

Get rid of blanks with `StringBuffer`

`StringBuffer` is empty



3) Everything in between is the first word. Extract it and append it to `StringBuffer` which is now “`Get`”

4) Append a blank and repeat the process!!

Example

```
public String function(String my_string)
{
      boolean flag = true, first_flag = true;
      StringBuffer buf;
      int index_start = 0, index_end;
      buf = new StringBuffer(50);
      while (flag)
      {
            if (first_flag)
                  first_flag = false;
            else
                  buf.append(' ');
      while(my_string.substring(index_start,index_start + 1)
            .equals(" "))
            index_start++;
      index_end = my_string.indexOf(" ", index_start);
```

Example

```
    if (index_end == -1)
    {
       buf.append(my_string.substring(index_start));
       flag = false;
       }
     else
     {
       buf.append(my_string.substring(index_start,
       index_end));
       index_start = index_end + 1;
     }
  }
  return buf.toString();
}
```

There are some bugs here… can you track them?

The `StringTokenizer` Class

- **`String`**s often consist of tokens that may have some meaning such as
  - individual words or perhaps keywords, identifiers, operators and other elements of a programming language
- Tokens are separated from one another by delimiters
  - whitespace characters such as blank, tab, newline and carriage return
  - they may include commas for CSV files and <> for HTML files
- The **`StringTokenizer`** class allows a string to broken down into its component tokens (part of the **`java.util.*`** package)
- The **`StringTokenizer`** class allows for the string to be tokenised to be specified as well as the delimiters and whether the delimiters are to be also returned as tokens
  - **`StringTokenizer(String s) // uses a default delimiter of " \n\t\r" consisting a space, a newline, a tab and a carriage return`**
  - **`StringTokenizer(String s, String delimiters);`**
  - **`StringTokenizer(String s, String delimiters, boolean returnTokens);`**

The `StringTokenizer` Class

- Returning the delimiters as tokens is useful if for instance commas separated fields and carriage returns separated records
- Using the delimiter you can determine whether the next token is a new field or a new record
- The number of tokens in a string can be determined by calling the **`countTokens()`** method
  - **`int StringTokenizer.countTokens()`**
- The **StringTokenizer** can also be used in a while loop using the **hasMoreTokens** and **nextToken** methods

```
while(tokens.hasMoreTokens())
{
   System.out.println(tokens.nextToken());
}
```

- Example : `((a_12 + 23.59) * (4563 - 32));`
  - If numbers and variable names are our tokens, the delimiters are characters in the string "()+*; "

Points to remember

- The constructor that we study is `StringTokenizer` It takes two strings:
  - String1 is the string to be processed
  - String2 contains the delimiters
  - other options are possible

- Once we construct an object of class `StringTokenizer`, we can use two methods:
  - `countTokens()` to tell us how many tokens we have
  - `nextToken()` to give us the next token

Squeezeblanks using tokenizer

1. Tokenize the string

2. Create an object `O` of type `StringBuffer`;

3.
```
While (there are more words in tokenizer)
{
    get the next word W from tokenizer;
    if (O already contains 1 or more words)
            append a space to O;
    append W to O;
}
```

Example

Consider the string - "get rid of extra blanks“

Step ❶ gives us the following tokens :

```
get     rid     of      extra    blanks
```

Step ❷ gives us an object O with a blank string ""

Step ❸ Successively gives us :

```
"get”  ➡  "get rid" ➡  "get rid of" ➡

"get rid of extra" ➡   "get rid of extra blanks "
```

Example

```
public String function(String my_string)
{
      int index, num_tokens;
      boolean first_flag = true;
      StringBuffer buf;
      StringTokenizer tokens =
            new StringTokenizer(my_string, " ");
      buf = new StringBuffer(50);
      num_tokens =  tokens.countTokens();
      for (index = 0; index < num_tokens; index++)
      {
            if (first_flag) first_flag = false;
            else buf.append(" ");
         buf.append(tokens.nextToken());
      }
      return buf.toString();
   }
```