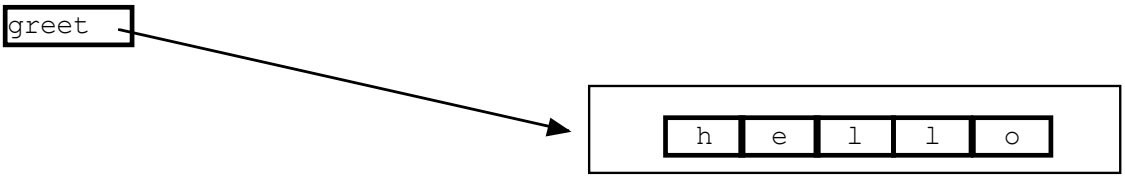
Strings and Chars

The **char** type (pronounced "car") represents a single character. A char literal value can be written in the code using single quotes (') like 'A' or 'a' or '6'. Java supports the **unicode** character set, so the characters can also be non-roman letters such as 'ø' or 'π'. The char type is a primitive, like int, so we use `==` and `!=` to compare chars. Rather than dealing with individual characters, we mostly deal with sequences of characters called **strings**.

A **string** is a sequence of characters, such as the string "Hello" or the string "What hath god wrought". A string could store a single word or the text of an entire book.

Java's powerful built-in **String** class provides great support for string operations. Each String object stores a sequence of chars, such as "Hello", and responds to methods that operate on those chars. We can create a String object the usual way with the **new** operator...

```
String greet = new String("hello");
```



String object

String objects are so common that there is a shorthand literal syntax to create them. We can create a String object simply by writing the desired text within double quotes "like this". The shorter version of the code below creates a new String object exactly as above.

```
String greet = "hello";
```

It's valid to create a string made of zero characters – the quotes are right next to each other with nothing in between. This is known as the **empty string**. The empty string is a valid string object, and the many operations described below work correctly on the empty string where it makes sense.

```
String empty = "";  // creates an empty string (zero chars)
```

Special Characters

What if we want to create a string with a double quote (") in it? Java uses backslash "escape" codes to insert special chars just as they are, such as \" to get a double quote character. Many computer languages use these same special backslash escape codes.

`\"` – a double quote char

`\\` – a backslash char

`\t` – a tab char

`\n` – a newline char (the common end-of-line char, like the return key on the keyboard)

`\r` – a carriage return (a less common end-of-line char)

String Concatenation +

When used between two int values, the + operator does addition. When used between two or more strings, the + operator appends them all together to form a new string.

```
String start = "Well";
String end = "hungry";

String result = start + ", I'm " + end; // result is "Well, I'm hungry!"
```

In computer science, the verb "concatenate" is used for this situation – appending sequences together to make one big sequence.

This special feature of the + operator also works when a string is combined with a primitive such as an int or double or char. The int or double value is automatically converted to string form, and then the strings are appended together. This feature of the + is very a convenient way to mix int and double values into a string...

```
int count = 123;
char end = '!';

String result = "I ate " + count + " donuts" + end;  // "I ate 123 donuts!"
```

String Immutable Style

String objects use the "immutable" style, which means that once a string object is created, it never changes. For example, notice that the above uses of + concatenation do not change existing strings. Instead, the + creates new strings, leaving the originals intact. As we study the many methods that strings respond to below, notice that no method changes an existing string. The immutable style is popular since it keeps things simple.

String Comparison

If we have two string objects, we use the equals() method to check if they are the same. In Java, we always use the equals() message to compare two objects – strings are just an example of that rule. For example, we used equals() to compare Color objects earlier in the quarter. The == operator is similar but is used to compare primitives such as int and char. Use equals() for objects, use == for primitives.

Unfortunately, using == with string objects will compile fine, but at runtime it will not actually compare the strings. It's easy to accidentally type in == with objects, so it is a common error.

The equals() method follows the pointers to the two String objects and compares them char by char to see if they are the same. This is sometimes called a "deep" comparison – following the pointers and looking at the actual objects. The comparison is "case-sensitive" meaning that the char 'A' is considered to be different from the char 'a'.

```
String a = "hello";
String b = "there";

// Correct -- use the .equals() method
if (a.equals("hello")) {
   System.out.println("a is \"hello\"");
}

// NO NO NO -- do not use == with Strings
if (a == "hello") {
   System.out.println("oops");
}


// a.equals(b) -> false
// b.equals("there") -> true
// b.equals("There") -> false
// b.equalsIgnoreCase("THERE") -> true
```

There is a variant of equals() called equalsIgnoreCase() that compares two strings, ignoring uppercase/lowercase differences.

String Methods

The Java String class implement many useful methods.

`int length()` – returns the number of chars in the receiver string. The empty string "" returns a length of 0.

```
String a = "Hello";
int len = a.length();  // len is 5
```

`String toLowerCase()` – returns a new string which is a lowercase copy of the receiver string. Does not change the receiver.

```
String a = "Hello";
String b = a.toLowerCase();  // b is "hello"
```

`String toUpperCase()` – like toLowerCase(), but returns an all uppercase copy of the receiver string. Does not change the receiver.

```
String a = "Hello";
String b = a.toUpperCase();  // b is "HELLO"
```

`String trim()` – returns a copy of the receiver but with whitespace chars (space, tab, newline, …) removed from the start and end of the String. Does not remove whitespace everywhere, just at the ends. Does not change the receiver.

```
String a = "   Hello   There   ";
String b = a.trim();  // b is "Hello There"
```

Using String Methods

Methods like trim() and toLowerCase() always return new string objects to use; they never change the receiver string object (this is the core of immutable style – the receiver object never changes). For example, the code below does not work...

```
String word = "  hello  ";
word.trim();  // ERROR, this does not change word

// word is still "  hello  ";
```

When calling trim() (or any other string method) we must use the **result returned** by the method, assigning into a new variable with = for example...

```
String word = "  hello  ";
String trimmed = word.trim();  // Ok, trimmed is "hello"
```

If we do not care about keeping the old value of the string, we can use = to assign the new value right into the old variable...

```
String word = "  hello  ";
word = word.trim();  // Ok, word is "hello" (after the assignment)
```

This works fine. The trim() method returns a new string ("hello") and we store it into our variable, forgetting about the old string.

Garbage Collector – GC

This is a classic case where the "Garbage Collector" (GC) comes in to action. The GC notices when heap memory is no longer being used, the original " hello " string in this case, and recycles the memory automatically. The languages C and C++ do not have built-in GC, and so code written in C/C++ must manually manage the creation and recycling of memory, which make the code more work to build and debug. The downside of GC is that it imposes a small cost, say around 10%, on the speed of the program. The cost of the GC is highly variable and depends very much on the particular program. As computers have gotten faster, and we want to build larger and more complex programs, GC has become a more and more attractive feature. Modern programmer efficient languages like Java and Python have GC.

String Comparison vs. Case

Suppose a program wants to allow the user to type in a word with any capitalization they desire ("summer" or "Summer" or "SUMMER"). How can the program check if the user

typed in the word "summer"? There are two reasonable strategies. One strategy is to use equalsIgnoreCase()...

```
String word = <typed by user>

if (word.equalsIgnoreCase("summer")) { ...  // this works
```

Another strategy is to use toLowerCase() (above) once to create a lowercase copy of the word, and then knowing that it is lowercase, just use equals() for comparisons...

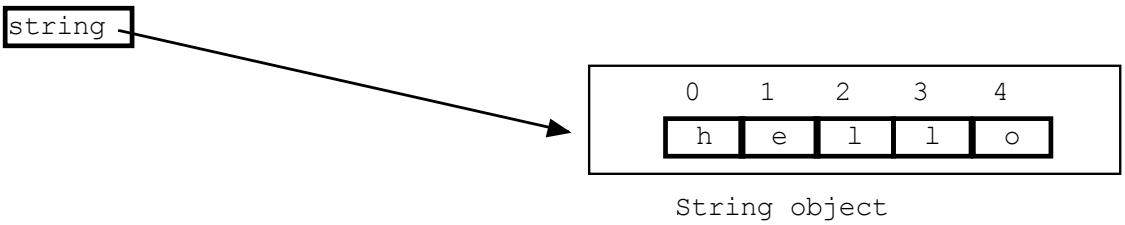
```
String word = <typed by user>

word = word.toLowerCase();        // convert to lowercase

if (word.equals("summer")) { ...  // now can just use equals()
```

String Indexing

The chars in a String are each identified by an index number, from 0 .. length-1. The leftmost char is at index 0, the next at index 1, and so on.



String object

This "zero-based" numbering style is pervasive in computer science for identifying elements in a collection, since it makes many common cases a little simpler. The method charAt(int index) returns an individual char from inside a string. The valid index numbers are in the range 0..length-1. Using an index number outside of that range will raise a runtime exception and stop the program at that point.

```
String string = "hello";
char a = string.charAt(0);  // a is 'h'
char b = string.charAt(4);  // b is 'o'
char c = string.charAt(string.length() - 1);  // same as above line
char d = string.charAt(99);  // ERROR, index out of bounds
```

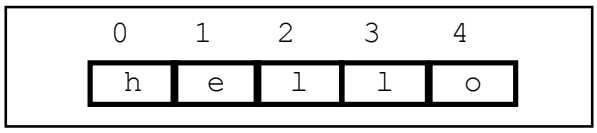
Substring

The substring() method is a way of picking out a sub-part of string. It uses index numbers to identify which parts of the string we want.

The simplest version of substring() takes a single int index value and returns a new string made of the chars starting at that index...

```
String string = "hello";
String a = string.substring(2);  // a is "llo"
String b = string.substring(3);  // b is "lo"
```

A more complex version of substring() takes both **start** and **end** index numbers, and returns a string of the chars between start and **one before** end...




```
String string = "hello";
String a = string.substring(2, 4);  // a is "ll" (not "llo")
String b = string.substring(0, 3);  // b is "hel"
```

Remember that substring() stops one short of the end index, as if the end index represents the start of the **next** string to get. In any case, it's easy to make off-by-one errors with index numbers in any algorithm. Make a little drawing of a string and its index numbers to figure out the right code.

As usual, substring() leaves the receiver string unchanged and returns its result in a new string. Therefore, code that calls substring() must capture the result of the call, as above.

String Method Chaining

It's possible to write a series of string methods together in a big chain. The methods evaluate from left to right, with each method using the value returned by the method on its left....

```
String a ="hello    There  ";
String b = a.substring(5).trim().toLowerCase(); // b is "there"
```

This technique works with Java methods generally, but it works especially well with string methods since most (substring, trim, etc.) return a string as their result, and so provide a string for the next method to the right to work on.

indexOf()

The indexOf() method searches inside the receiver string for a "target" string. IndexOf() returns the index number where the target string is first found (searching left to right), or -1 if the target is not found.

`int indexOf(String target)` – searches for the target string in the receiver. Returns the index where the target is found, or -1 if not found.

The search is case-sensitive – upper and lowercase letters must match exactly.