

Using Classes and objects

Dr. 何明昕, He Mingxin, Max

program06 @ yeah.net

Email Subject: (AE | A2 | A3) + *(Last 4 digits of ID) + Name: TOPIC*

Sakai: CS102A in 2018A

计算机程序设计基础

Introduction to Computer Programming

Using Classes and objects

The ArrayLists

Ch07, 7.14, Java™ How to Program

Object Oriented Thinking

Ch10, Liang's Book

Contents

❑ Basics of Classes and Objects

❑ ArrayLists (§7.14, jhttp9)

❑ To create objects for primitive values using the wrapper classes (**Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Character**, and **Boolean**) (§10.7).

❑ To simplify programming using automatic conversion between primitive types and wrapper class types (§10.8).

❑ To use the **BigInteger** and **BigDecimal** classes for computing very large numbers with arbitrary precisions (§10.9).

❑ To use the **String** class to process immutable strings (§10.10).

❑ To use the **StringBuilder** and **StringBuffer** classes to process mutable strings (§10.11).

❑ **Regular Expression** (正则表达式) , String Matching & Splitting

Software Engineering as Managing Change

The computer doesn't care how code is organized – but humans care a lot.

Changing code is hard and expensive, and because the world changes, essential. We want to make it easy and cheap, by:

- ✓ minimizing the amount of code that must change
- ✓ it easy to work out which code must change
- ✓ having the code that must change live together (changing 6 lines in one file is typically cheaper than changing 1 line of code in each of 6 files).

Criteria for Good Programs:

- 1) Run Effectively (Correctly) & Efficiently (Objectives 目的)
- 2) Easy to be Extended and Modified (Approaches 手段)
- 3) Easy to be Understood (Pre-conditions 前提)

How can we make change easier and cheaper?

Key idea: **hide** certain information inside well-defined pieces of code, so that users of that piece of code don't depend on it, and don't need to change if it changes.

E.g. **caller of a function doesn't know how the function works:**
just what argument types it takes and what type it returns.

This was an innovation in its time! (Look up “GOTO considered harmful”.)

But what else should be hidden?

Data representation

Recall: a **data type** is a **set of values** (with limited storage and specific formats) and **operations on those values**. May be

➤ **primitive**, built into the language with operations defined in the compiler/runtime, e.g. **int**, **double**, **boolean** or

➤ **user-defined**, with operations defined in the programming language itself, e.g. **PrinterQueue**, **HotelRoom**, ...

In Java, **String** is an interesting intermediate case: you may have thought of it as primitive, but it's defined in Java (google java **String.java** to see how). And then there are the standard libraries.

Hiding data representation (Information Hidden)

You shouldn't need to know how a data type is implemented in order to use it. It should suffice to read the documentation: what operations are there, what do they do?

Then you **can** write code that won't need to change if the implementation changes. (Whether you **may** write code that does depend on the implementation is another matter. If not, the data type is **encapsulated (封装)**.)

The general idea is not specific to OO, but Java has its own OO models.

Towards Object Oriented Programming...

So far in this course, we've been doing **Procedural Programming** [verb oriented]

- tell the computer to do this, then
- tell the computer to do that.

You know:

- ✓ how to program with primitive data types e.g. **int**, **boolean**;
- ✓ how to control program flow to do things with them, e.g. using **if**, **for**;
- ✓ how to group similar data into arrays.

You've glimpsed **OO**: e.g., all our code is organized inside classes, and you have used **Strings**.

Philosophy of Object Orientation

The problem with structuring your software based on what it must do is that that changes a lot.

The **domain** in which it works changes much less.

Key insight: structuring your software around the **things** in the domain makes it easier to understand and maintain.

(Many key names: look up Alan Kay, Rebecca Wirfs-Brock, Grady Booch, Barabara Liskov, Bertrand Meyer for writing on all this...)

Object Oriented programming (OOP) [**noun oriented**]

➤ Things in the world **know** things: instance variables.

➤ Things in the world **do** things: methods.

In other words, **objects have state and behavior.**

Intuition



Client

API

Implementation

- ▶ adjust volume
- ▶ switch channel
- ▶ switch to standby

- ▶ cathode ray tube
- ▶ 20" screen, 22 kg
- ▶ Sony Trinitron KV20M10

client needs to know
how to use API

implementation needs
to know what API to
implement

Implementation and client need to agree on API ahead of time.

Intuition



Client

API

Implementation

- ▶ adjust volume
- ▶ switch channel
- ▶ switch to standby

- ▶ HD LED display
- ▶ 37" screen, 10 kg
- ▶ Samsung UE37C5800

client needs to know
how to use API

implementation needs
to know what API to
implement

Can substitute better implementation without changing the client.

TV Example

Using the TV API

```
TV mytv = new TV();  
mytv.setVolume( up );  
mytv.setChannel( Film4 );  
mytv.standBy();
```

Note that we have two independent ideas here:

- ✓ Conceptual objects such as `mytv` are directly present in the program;
I
- ✓ They have static (compile-time) types that define their behavior.

Classes: user-defined data types

- ▶ Java is a class-based object-oriented language.
- ▶ A class is a kind of data type that you can define yourself.
- ▶ An object is an instance of a class.
- ▶ The code in a Java system is organized into classes.
- ▶ The system runs by objects sending messages to one another, and reacting to receiving messages, possibly by sending other messages...
the sender trusts the receiver to do the right thing
- ▶ (Most) variables refer to objects.

Java also comes with a lot of standardized classes, such as `String`. Let's look at how `String` fits into this picture.

Constructors and Methods

declare a variable (object name)



call a constructor to create an object

String s;



s = new String("Hello, World!");

System.out.println(s.substring(0, 5));

object name



call an instance method that operates on the object's value



For short:

```
s = "Hello, World!" ;
```

```
// There is some difference indeed!
```

Static Methods vs. Instance Methods

Static Methods (Class Methods):

- ▶ Associated with a **class**.
- ▶ Identifying a method in a separate class requires name of the class:

`Math.abs()`, `Gaussian.pdf()`.

Instance Methods (Object Methods):

- ▶ Associated with an **object**.
- ▶ Identifying an instance method requires an object name:

`s.substring()`

String: basis for text processing

Underlying **set of values**: sequences of Unicode characters.

In Java **Strings** are **immutable**: none of the operations change the value.

```
public class String
```

<code>String(String s)</code>	<i>create a string with same value as s</i>
<code>char charAt(int i)</code>	<i>character at index i</i>
<code>String concat(String t)</code>	<i>this string with t appended</i>
<code>int compareTo(String t)</code>	<i>compare lexicographically with t</i>
<code>boolean endsWith(String post)</code>	<i>does string end with post?</i>
<code>boolean equals(Object t)</code>	<i>is t a String equal to this one?</i>
<code>int indexOf(String p)</code>	<i>index of first occurrence of p</i>
<code>int indexOf(String p, int i)</code>	<i>as indexOf, starting search at index i</i>
<code>int length()</code>	<i>return length of string</i>
<code>String replaceAll(String a, String b)</code>	<i>result of changing all as to bs</i>
<code>String[] split(String delim)</code>	<i>result of splitting string at delim</i>
<code>boolean startsWith(String pre)</code>	<i>does string start with pre?</i>
<code>String substring(int i, int j)</code>	<i>from index i to index j - 1 inclusive</i>

<http://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

Typical String Processing Code

is the string a palindrome?

```
public static boolean isPalindrome(String s) {  
    int N = s.length();  
    for (int i = 0; i < N / 2; i++) {  
        if (s.charAt(i) != s.charAt(N - 1 - i))  
            return false;  
    }  
    return true;  
}
```

*extract file names and extensions
from a command-line argument*

```
String s = args[0];  
int dot = s.indexOf(".");  
String base = s.substring(0, dot);  
String extension = s.substring(dot + 1, s.length());
```

*print all lines from standard input
containing the string "info"*

```
while (input.hasNext()) {  
    String s = input.nextLine();  
    if (s.contains("info"))  
        System.out.println(s);  
}
```

*print all sust.edu.cn URLs
in text file on standard output*

```
while (input.hasNext()) {  
    String s = input.nextLine();  
    if (s.startsWith("http://") && s.endsWith("sust.edu.cn"))  
        System.out.println(s);  
}
```

Strings and Equality

`equals()`: are the characters inside a `String` object the same?

`==`: do the two object references refer to the same instance?

In all the coursework for this course, you should be using `equals()` for checking whether two strings are 'the same'.

String client example: gene finding

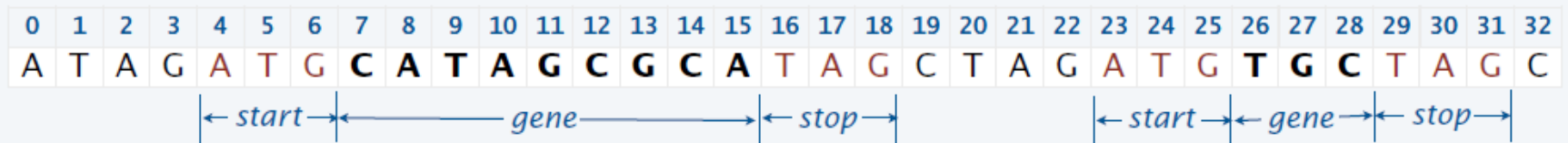
Pre-genomics era. Sequence a human genome.

Post-genomics era. Analyze the data and understand structure.

Genomics. Represent genome as a string over A C T G alphabet.

Gene. A substring of genome that represents a functional unit.

- Made of *codons* (three A C T G *nucleotides*).
- Preceded by ATG (*start* codon).
- Succeeded by TAG, TAA, or TGA (*stop* codon).



Goal. Write a Java program to find genes in a given genome.

String client warmup: Identifying a potential gene

Goal. Write a Java program to determine whether a given string is a potential gene.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	T	G	C	A	T	A	G	C	G	C	A	T	A	G

← *start* → ← *gene* → ← *stop* →

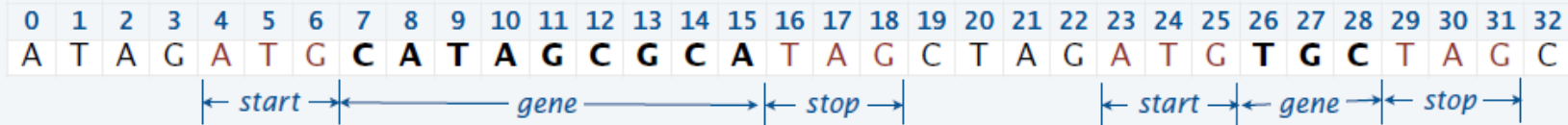
```
% java Gene ATGCATAGCGCATAG
true
% java Gene ATGCGCTGCGTCTGTACTAG
false
% java Gene ATGCCGTGACGTCTGTACTAG
false
```

```
public class Gene
{
    public static boolean isPotentialGene(String dna)
    {
        if (dna.length() % 3 != 0) return false;
        if (!dna.startsWith("ATG")) return false;
        for (int i = 0; i < dna.length() - 3; i+=3)
        {
            String codon = dna.substring(i, i+3);
            if (codon.equals("TAA")) return false;
            if (codon.equals("TAG")) return false;
            if (codon.equals("TGA")) return false;
        }
        if (dna.endsWith("TAA")) return true;
        if (dna.endsWith("TAG")) return true;
        if (dna.endsWith("TGA")) return true;
        return false;
    }
    public static void main(String[] args)
    {
        StdOut.println(isPotentialGene(args[0]));
    }
}
```

Exercise Six (Oct. 23), Last submitting Date: Nov. 1.

String client exercise: Gene finding

Goal. Write a Java program to find genes in a given genome.



Algorithm. Scan left-to-right through dna.

- If start codon ATG found, set **beg** to index *i*.
- If stop codon found and substring length is a multiple of 3, print gene and reset **beg** to -1.

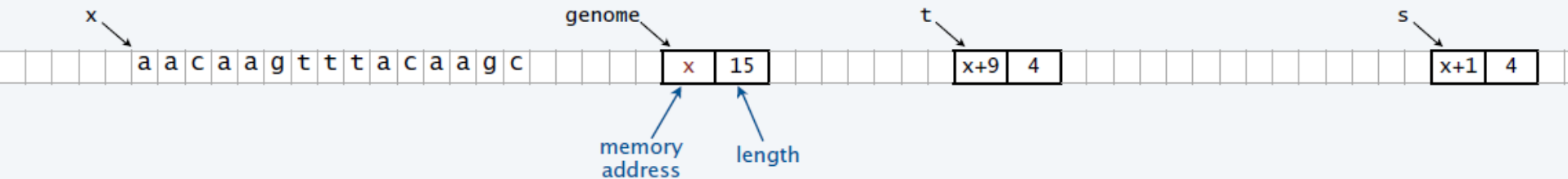
i	codon		beg	output	remainder of input string
	start	stop			
0			-1		ATAGATGCATAGCGCATAGCTAGATGTGCTAGC
1		TAG	-1		TAGATGCATAGCGCATAGCTAGATGTGCTAGC
4	ATG		4		ATGCATAGCGCATAGCTAGATGTGCTAGC
9		TAG	4		TAGCGCATAGCTAGATGTGCTAGC
16		TAG	4	CATAGCGCA	TAGCTAGATGTGCTAGC
20		TAG	-1		TAGATGTGCTAGC
23	ATG		23		ATGTGCTAGC
29		TAG	23	TGC	TAGC

Implementation. Entertaining programming exercise!

OOP context for strings

Possible memory representation of

```
String genome = "aacaagtttacaagc";  
String s = genome.substring(1, 5);  
String t = genome.substring(9, 13);
```



Implications

- `s` and `t` are different strings that share the same value "acaa".
- `(s == t)` is false (because it compares addresses).
- `(s.equals(t))` is true (because it compares character sequences).
- Java String interface is more complicated than the API.

A homemade class: Circle

Next time, we'll see how to define a `Circle` class (in several variants). Let's start by seeing how we might use one. Suppose its API is:

```
public class Circle
    Circle(double radius)    constructor
    double  getArea()
    void    enlarge(int scaleFactor)
    boolean equals(Object o)    true iff o is a Circle of same size
```

Unlike `String`, `Circle` is mutable: its state can be changed by sending it message `enlarge`.

Using Circle

```
Circle c1 = new Circle(1);  
double a1 = c1.getArea(); // pi
```

```
Circle c2 = new Circle(2);  
double a2 = c2.getArea(); // 4 pi
```

```
Circle c3 = c1; // two references to same object  
double a3 = c3.getArea(); // pi
```

```
System.out.println (c1 == c2); // false  
System.out.println (c1.equals(c2)); // also false
```

```
System.out.println (c1 == c3); // true  
System.out.println (c1.equals(c3)); // also true
```

Using Circle, continued

```
c1.enlarge(2);

double a1new = c1.getArea(); // now 4 pi
double a2new = c2.getArea(); // still 4 pi
double a3new = c3.getArea(); // now 4 pi

System.out.println (c1 == c2); // still false
System.out.println (c1.equals(c2)); // now true

System.out.println (c1 == c3); // still true
System.out.println (c1.equals(c3)); // also still true
```

A word about new

Local* variables of primitive type use stack space and are created like this:

```
public void foo() {  
    int i = 5;  
    //...  
}
```

Things of reference type use heap space† and are created with **new**, which allocates the right amount of space and, for objects, calls a constructor. E.g.

```
public void bar() {  
    int[] a = new int[5]; // allocate space for 5 ints  
    Circle c = new Circle(2); // allocate space for a Circle  
    //...  
}
```

Possible sources of confusion

1. * Primitively-typed attributes of objects (e.g. the radius of the circle), or primitively-typed elements of arrays (e.g. the contents of the `int` array), are on the **heap** (堆) with their owning object. (Otherwise putting the owning object on the heap would be pointless: it would collapse when its innards went out of scope!)
2. † a local reference to a thing of reference type lives on the **Stack** (栈), even though the thing itself is in the heap: once method bar the stack memory containing references a and c is reclaimed.

If there are other references to the same heap objects elsewhere in the program – e.g., a or c have been passed to a method still in progress, or stored as attributes of an object – those references can still be used.

If there are no other references, the heap objects will now be available for garbage collection (GC).

Summary: Object

An object has state, behavior and identity.

- ▶ **State**: the data that the object **encapsulates**
the object may decide to modify its state in response to a message
but **immutable** objects never do this
- ▶ **Behavior**: the messages the object understands, and what it does in response
exactly what an object does in response to a message may depend on its state
- ▶ **Identity**: two objects may currently have identical data, but be different objects
e.g. o1 and o2 could be in the same state now, but not after you sent a message to o1, which affects o1's state but not o2's.

Summary: Why use object orientation?

OO has taken the world by storm. Why?

For *programming*, OOP is nice, but not clearly better than FP, say.

For *software engineering*, OO is better for most applications.

- ▶ use objects to model real-world things
- ▶ use classes to model **domain concepts**.
- ▶ These change more slowly than specific functional requirements.
- ▶ so what OO does is to **put things together that change together** as requirements evolve.

Change is the thing that makes software engineering hard and interesting; OO helps manage it.

Summary: in Java

A variable can have

- ▶ a **primitive type** e.g., `boolean`, `int`, `double`; or
- ▶ a **reference type**: **any class**, e.g. `String`, `Picture`, `Color`, **any array type**.

Things of reference type are created using `new` and destroyed by being garbage collected automatically.

Variables of reference type contain references to those things, not the things themselves. So:

- ▶ Two references can refer to the same thing.
- ▶ Copying the reference does not copy the thing, so...
- ▶ ... when you pass the reference into a method, and the method uses the reference to change the state of the thing, the change is visible outside.
- ▶ Need to distinguish `==` (these references refer to the same thing) from `.equals` (these references refer to things that are currently equal).

Object-oriented programming: summary

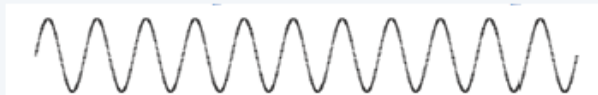
Object-oriented programming.

- Create your own data types (sets of values and ops on them).
- Use them in your programs (manipulate *objects*).

← An **object** holds a data type value.
Variable names refer to objects.

In Java, programs manipulate references to objects.

- String, Picture, Color, arrays, (and everything else) are *reference types*.
- Exceptions: boolean, int, double and other *primitive types*.
- OOP purist: Languages should not have separate primitive types.
- Practical programmer: Primitive types provide needed efficiency.



T	A	G	A	T	G	T	G	C	T	A	G	C
---	---	---	---	---	---	---	---	---	---	---	---	---

This lecture: You can write programs to manipulate strings and others.

Next lecture: You can *define your own abstractions* and write programs that manipulate them.

Object-oriented programming: summary

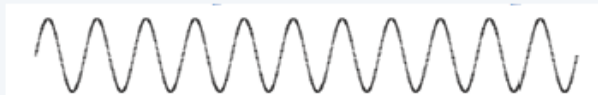
Object-oriented programming.

- Create your own data types (sets of values and ops on them).
- Use them in your programs (manipulate *objects*).

← An **object** holds a data type value.
Variable names refer to objects.

In Java, programs manipulate references to objects.

- String, Picture, Color, arrays, (and everything else) are *reference types*.
- Exceptions: boolean, int, double and other *primitive types*.
- OOP purist: Languages should not have separate primitive types.
- Practical programmer: Primitive types provide needed efficiency.



T	A	G	A	T	G	T	G	C	T	A	G	C
---	---	---	---	---	---	---	---	---	---	---	---	---

This lecture: You can write programs to manipulate strings and others.

Next lecture: You can *define your own abstractions* and write programs that manipulate them.

Introduction to Collections and Class ArrayList

Java API provides several predefined data structures, called **collections**, used to store groups of related objects.

Each provides efficient methods that **organize, store and retrieve your data** without requiring knowledge of how the data is being stored.

Reduce application-development time.

Arrays do not automatically change their size at execution time to accommodate additional elements.

ArrayList<T> (package java.util) can dynamically change its size to accommodate more elements.

T is a placeholder for the type of element stored in the collection.

This is similar to specifying the type when declaring an array, except that only non-primitive types can be used with these collection classes.

Classes with this kind of placeholder that can be used with any type are called **generic classes (泛型类)**.

Method	Description
<code>add</code>	Adds an element to the end of the <code>ArrayList</code> .
<code>clear</code>	Removes all the elements from the <code>ArrayList</code> .
<code>contains</code>	Returns <code>true</code> if the <code>ArrayList</code> contains the specified element; otherwise, returns <code>false</code> .
<code>get</code>	Returns the element at the specified index.
<code>indexOf</code>	Returns the index of the first occurrence of the specified element in the <code>ArrayList</code> .
<code>remove</code>	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
<code>size</code>	Returns the number of elements stored in the <code>ArrayList</code> .
<code>trimToSize</code>	Trims the capacity of the <code>ArrayList</code> to current number of elements.

Fig. 7.23 | Some methods and properties of class `ArrayList<T>`.

`Object[] toArray()`

`<T> T[] toArray(T[] a)`

...

Intro. to Collections and Class ArrayList (cont.)

ArrayList collection

- Similar to arrays

- Dynamic resizing

They automatically increase their size at execution time to accommodate additional elements.

An ArrayList's **capacity** indicates how many items it can hold without growing.

When the ArrayList grows, it must create a larger internal array and copy each element to the new array.

This is a time-consuming operation. It would be inefficient for the ArrayList to grow each time an element is added.

An ArrayList grows only when an element is added and the number of elements is equal to the capacity—i.e., there is no space for the new element.

Intro. to Collections and Class ArrayList (Cont.)

Method add adds elements to the ArrayList.

- One-argument version appends its argument to the end of the ArrayList.

- Two-argument version inserts a new element at the specified position.

- Collection indices start at zero.

Method size returns the number of elements in the ArrayList.

Method get obtains the element at a specified index.

Method remove deletes an element with a specific value.

- An overloaded version of the method removes the element at the specified index.

Method contains determines if an item is in the ArrayList.

```
1 // Fig. 7.24, jhttp9 (revised): ArrayListCollection.java
2 // Generic ArrayList collection demonstration.
3 import java.util.ArrayList;
4
5 public class ArrayListCollection {
6     public static void main (String[] args) {
7         // create a new ArrayList of Strings with an initial capacity of 10
8         ArrayList<String> items = new ArrayList<String>( 10 );
9
10        items.add( "red" );           // append an item to the list
11        items.add( 0, "yellow" );    // insert "yellow" at index 0
12
13        // header
14        System.out.print( "Display list contents with counter-controlled loop:" );
15
16        // display the colors in the list<input type="button" />
17        for (int i = 0; i < items.size(); i++)
18            System.out.printf( " %s", items.get(i) );
19
20        // display colors using enhanced for in the display method
21        display( items, "\nDisplay list contents with enhanced for statement:" );
22
23        items.add( "green" ); // add "green" to the end of the list
24        items.add( "yellow" ); // add "yellow" to the end of the list
25        display( items, "List with two new elements:" );
```

```
26
27 items.remove( "yellow" ); // remove the first "yellow"
28 display( items, "Remove first instance of yellow:" );
29
30 items.remove( 1 ); // remove item at index 1
31 display( items, "Remove second list element (green):" );
32
33 // check if a value is in the List
34 System.out.printf( "\"red\" is %sin the list\n",
35     items.contains("red") ? "" : "not "
36 );
37
38 // display number of elements in the List
39 System.out.printf( "Size: %d\n", items.size() );
40 }
41
42 // display the ArrayList's elements on the console
43 public static void display (ArrayList<String> items, String header) {
44     System.out.print( header ); // display header
45
46     // display each element in items
47     for (String item : items)
48         System.out.printf( " %s", item );
49
50     System.out.println();
51 }
52 }
```

```
H:\work\JavaProg\2018Spring\notes08>javac ArrayListCollection.java
```

```
H:\work\JavaProg\2018Spring\notes08>java ArrayListCollection
```

```
Display list contents with counter-controlled loop: yellow red
```

```
Display list contents with enhanced for statement: yellow red
```

```
List with two new elements: yellow red green yellow
```

```
Remove first instance of yellow: red green yellow
```

```
Remove second list element (green): red yellow
```

```
"red" is in the list
```

```
Size: 2
```


Wrapper Classes

❑ Boolean

❑ Character

❑ Short

❑ Byte

❑ Integer

❑ Long

❑ Float

❑ Double

NOTE:

- (1) The wrapper classes do not have no-arg constructors.
- (2) The instances of all wrapper classes are immutable, i.e., their internal values cannot be changed once the objects are created.

Integer and Double Classes

java.lang.Integer

-value: int

+MAX VALUE: int

+MIN VALUE: int

+Integer(value: int)

+Integer(s: String)

+byteValue(): byte

+shortValue(): short

+intValue(): int

+longVlaue(): long

+floatValue(): float

+doubleValue():double

+compareTo(o: Integer): int

+toString(): String

+valueOf(s: String): Integer

+valueOf(s: String, radix: int): Integer

+parseInt(s: String): int

+parseInt(s: String, radix: int): int

java.lang.Double

-value: double

+MAX VALUE: double

+MIN VALUE: double

+Double(value: double)

+Double(s: String)

+byteValue(): byte

+shortValue(): short

+intValue(): int

+longVlaue(): long

+floatValue(): float

+doubleValue():double

+compareTo(o: Double): int

+toString(): String

+valueOf(s: String): Double

+valueOf(s: String, radix: int): Double

+parseDouble(s: String): double

+parseDouble(s: String, radix: int): double

Integer and Double Classes

- ❑ Constructors
- ❑ Class Constants MAX_VALUE, MIN_VALUE
- ❑ Conversion Methods

Numeric Wrapper Class Constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value. The constructors for Integer and Double are:

```
public Integer(int value)
```

```
public Integer(String s)
```

```
public Double(double value)
```

```
public Double(String s)
```

Numeric Wrapper Class Constants

Each numerical wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`. `MAX_VALUE` represents the maximum value of the corresponding primitive data type.

For `Byte`, `Short`, `Integer`, and `Long`, `MIN_VALUE` represents the minimum byte, short, int, and long values.

For `Float` and `Double`, `MIN_VALUE` represents the minimum *positive* float and double values.

The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E-45), and the maximum double floating-point number (1.79769313486231570e+308d).

Conversion Methods

Each numeric wrapper class implements the abstract methods doubleValue, floatValue, intValue, longValue, and shortValue, which are defined in the Number class. These methods “convert” objects into primitive type values.

The Static valueOf Methods

The numeric wrapper classes have a useful class method, `valueOf(String s)`.

This method creates a new object initialized to the value represented by the specified string. For example:

```
Double doubleObject = Double.valueOf( "12.4" );
```

```
Integer integerObject = Integer.valueOf( "12" );
```

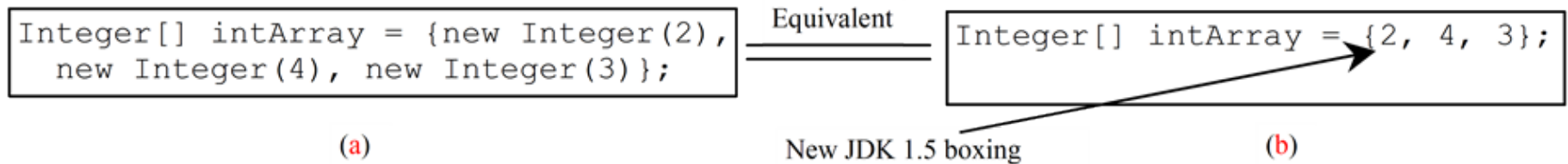
The Methods for Parsing Strings into Numbers

You have used the `parseInt` method in the `Integer` class to parse a numeric string into an int value and the `parseDouble` method in the `Double` class to parse a numeric string into a double value.

Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

Automatic Conversion Between Primitive Types and Wrapper Class Types

JDK 1.5 allows primitive type and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b):



Integer[] intArray = { 1, 2, 3 };
System.out.println(intArray[0] + intArray[1] + intArray[2]);

Unboxing

BigInteger and BigDecimal

If you need to compute with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes in the java.math package. Both are *immutable*.

Both extend the Number class and implement the Comparable interface.

BigInteger and BigDecimal

```
BigInteger a = new BigInteger( "9223372036854775807" );  
BigInteger b = new BigInteger( "2" );  
BigInteger c = a.multiply( b ); // 9223372036854775807 * 2  
System.out.println(c);
```

```
BigDecimal a = new BigDecimal( 1.0 );  
BigDecimal b = new BigDecimal( 3 );  
BigDecimal c = a.divide( b, 20, BigDecimal.ROUND_UP );  
System.out.println( c );
```

The String Class

☐ Constructing a String:

```
String message = "Welcome to Java";  
String message = new String("Welcome to Java");  
String s = new String();
```

☐ Obtaining String length and Retrieving Individual Characters in a String

☐ String Concatenation (concat)

☐ Substrings (substring(index), substring(start, end))

☐ Comparisons (equals, compareTo)

☐ String Conversions

☐ Finding a Character or a Substring in a String

☐ Conversions between Strings and Arrays

☐ Converting Characters and Numeric Values to Strings

Constructing Strings

```
String newString = new String(stringLiteral);
```

```
String message = new String("Welcome to Java");
```

Since strings are used frequently, Java provides a shorthand initializer for creating a string:

```
String message = "Welcome to Java";
```

Strings Are Immutable

A String object is immutable; its contents cannot be changed.

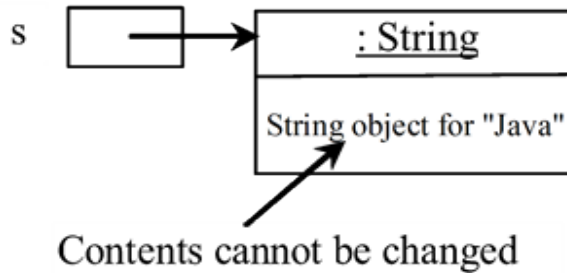
Does the following code change the contents of the string?

```
String s = "Java";  
s = "HTML";
```

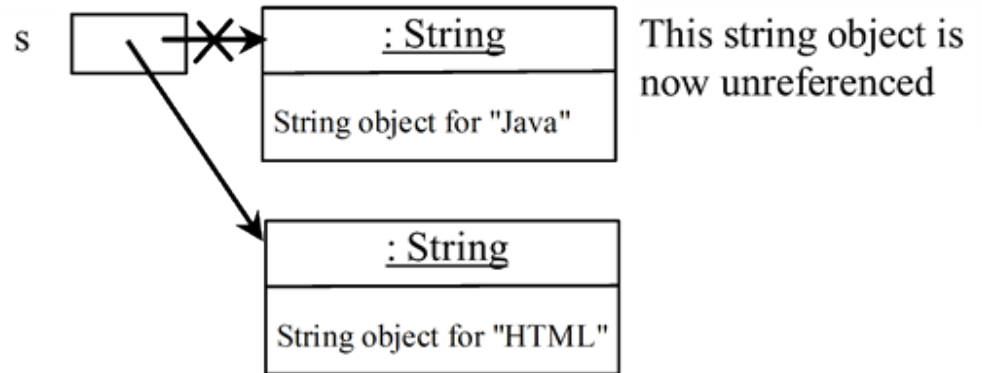
Trace Code

```
String s = "Java";  
s = "HTML";
```

After executing `String s = "Java";`



After executing `s = "HTML";`



Interned Strings

Since strings are immutable and are frequently used, to improve efficiency and save memory, the JVM uses a unique instance for string literals with the same character sequence. Such an instance is called *interned* (集中营, 多路共用的).

For example, the following statements in the next slide:

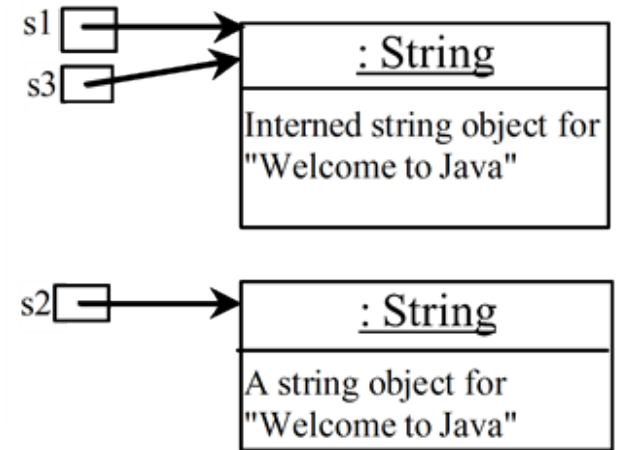
Examples

```
String s1 = "Welcome to Java";
```

```
String s2 = new String("Welcome to Java");
```

```
String s3 = "Welcome to Java";
```

```
System.out.println("s1 == s2 is " + (s1 == s2));  
System.out.println("s1 == s3 is " + (s1 == s3));
```



display

s1 == s is false

s1 == s3 is true

A new object is created if you use the new operator.

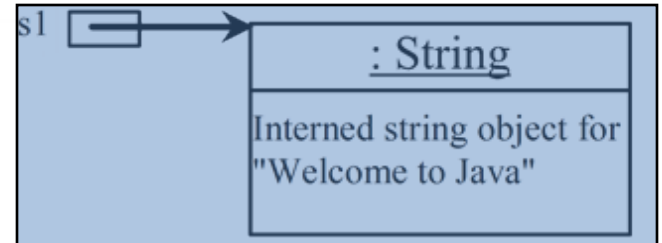
If you use the String initializer, no new object is created if the interned object is already created.

Trace Code

```
String s1 = "Welcome to Java";
```

```
String s2 = new String("Welcome to Java");
```

```
String s3 = "Welcome to Java";
```

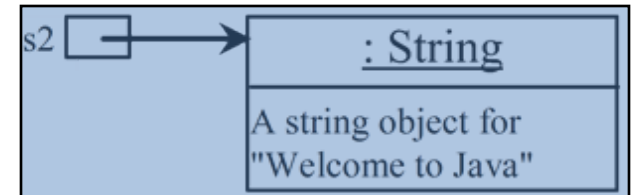
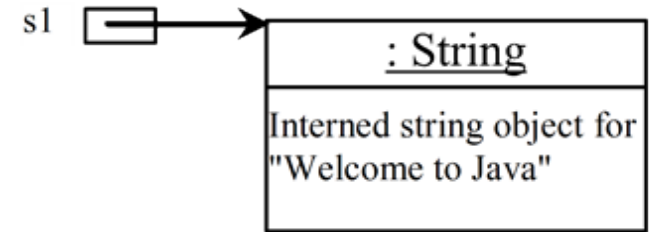


Trace Code

```
String s1 = "Welcome to Java";
```

```
String s2 = new String("Welcome to Java");
```

```
String s3 = "Welcome to Java";
```

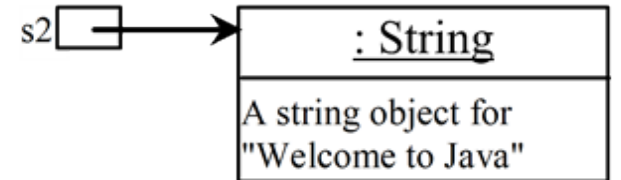
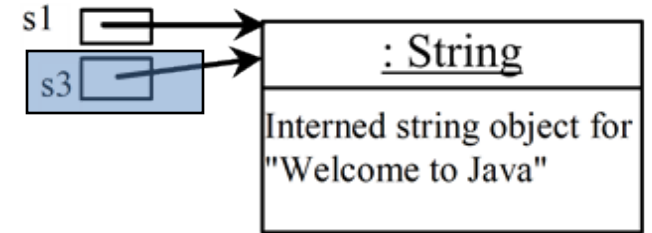


Trace Code

```
String s1 = "Welcome to Java";
```

```
String s2 = new String("Welcome to Java");
```

```
String s3 = "Welcome to Java";
```



Replacing and Splitting Strings

java.lang.String	
+replace(oldChar: char, newChar: char): String	Returns a new string that replaces all matching character in this string with the new character.
+replaceFirst(oldString: String, newString: String): String	Returns a new string that replaces the first matching substring in this string with the new substring.
+replaceAll(oldString: String, newString: String): String	Returns a new string that replace all matching substrings in this string with the new substring.
+split(delimiter: String): String[]	Returns an array of strings consisting of the substrings split by the delimiter.

Examples

`"Welcome".replace('e', 'A')` returns a new string, `WA1comA`.

`"Welcome".replaceFirst("e", "AB")` returns a new string, `WAB1come`.

`"Welcome".replace("e", "AB")` returns a new string, `WAB1comAB`.


`"Welcome".replace("el", "AB")` returns a new string, `WABcome`.

Splitting a String

```
String[] tokens = "Java#HTML#Perl".split( "#", 0);  
for (int i = 0; i < tokens.length; i++)  
    System.out.print( tokens[i] + " ");
```

displays

Java HTML Perl



Limit of
return
values

Matching, Replacing and Splitting by Patterns

You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature, commonly known as *regular expression* (正则表达式).

Regular expression is complex to beginning students. For this reason, two simple patterns are used in this section. Please refer to Supplement III.F, “Regular Expressions,” for further studies.

```
"Java".matches("Java")
```

```
"Java".equals("Java")
```

```
"Java is fun".matches("Java.*")
```

```
"Java is cool".matches("Java.*")
```


Matching, Replacing and Splitting by Patterns

The `replaceAll`, `replaceFirst`, and `split` methods can be used with a regular expression. For example, the following statement returns a new string that replaces \$, +, or # in "a+b\$#c" by the string NNN.

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");  
System.out.println(s);
```

Here the regular expression `[$+#]` specifies a pattern that matches \$, +, or #. So, the output is `aNNNbNNNNNNc`.

Matching, Replacing and Splitting by Patterns

The following statement splits the string into an array of strings delimited by some punctuation marks.

```
String[] tokens = "Java,C?C#,C++".split( "[.,:;?]" );  
  
for (int i = 0; i < tokens.length; i++)  
    System.out.println( tokens[i] );
```

Outputs:

Java

C

C#

C++

Convert Character and Numbers to Strings

The String class provides several static `valueOf` methods for converting a character, an array of characters, and numeric values to strings. These methods have the same name `valueOf` with different argument types `char`, `char[]`, `double`, `long`, `int`, and `float`. For example, to convert a double value to a string, use

`String.valueOf(5.44)`.

The return value is string consists of characters '5', '.', '4', and '4'.

`5.44`

`"" + 5.44`

StringBuilder and StringBuffer

The `StringBuilder/StringBuffer` class is an alternative to the `String` class.

In general, a `StringBuilder/StringBuffer` can be used wherever a string is used. `StringBuilder/StringBuffer` is more flexible than `String`.

You can add, insert, or append new contents into a string buffer, whereas the value of a `String` object is fixed once the string is created.

A list of differences between `StringBuffer` and `StringBuilder` are given below:

No.	<code>StringBuffer</code>	<code>StringBuilder</code>
1)	<code>StringBuffer</code> is <i>synchronized</i> i.e. thread safe. It means two threads can't call the methods of <code>StringBuffer</code> simultaneously.	<code>StringBuilder</code> is <i>non-synchronized</i> i.e. not thread safe. It means two threads can call the methods of <code>StringBuilder</code> simultaneously.
2)	<code>StringBuffer</code> is <i>less efficient</i> than <code>StringBuilder</code> .	<code>StringBuilder</code> is <i>more efficient</i> than <code>StringBuffer</code> .

StringBuilder Constructors

java.lang.StringBuilder
+StringBuilder()
+StringBuilder(capacity: int)
+StringBuilder(s: String)

Constructs an empty string builder with capacity 16.

Constructs a string builder with the specified capacity.

Constructs a string builder with the specified string.

Modifying Strings in the StringBuilder

<code>+append(data: char[]): StringBuilder</code>	Appends a char array into this string builder.
<code>+append(data: char[], offset: int, len: int): StringBuilder</code>	Appends a subarray in data into this string builder.
<code>+append(v: <i>aPrimitiveType</i>): StringBuilder</code>	Appends a primitive type value as a string to this builder.
<code>+append(s: String): StringBuilder</code>	Appends a string to this string builder.
<code>+delete(startIndex: int, endIndex: int): StringBuilder</code>	Deletes characters from startIndex to endIndex.
<code>+deleteCharAt(index: int): StringBuilder</code>	Deletes a character at the specified index.
<code>+insert(index: int, data: char[], offset: int, len: int): StringBuilder</code>	Inserts a subarray of the data in the array to the builder at the specified index.
<code>+insert(offset: int, data: char[]): StringBuilder</code>	Inserts data into this builder at the position offset.
<code>+insert(offset: int, b: <i>aPrimitiveType</i>): StringBuilder</code>	Inserts a value converted to a string into this builder.
<code>+insert(offset: int, s: String): StringBuilder</code>	Inserts a string into this builder at the position offset.
<code>+replace(startIndex: int, endIndex: int, s: String): StringBuilder</code>	Replaces the characters in this builder from startIndex to endIndex with the specified string.
<code>+reverse(): StringBuilder</code>	Reverses the characters in the builder.
<code>+setCharAt(index: int, ch: char): void</code>	Sets a new character at the specified index in this builder.

Examples

`StringBuilder stringBuilder = new StringBuilder("Welcome to ");`
`stringBuilder.append("Java");` → **Welcome to Java**
`stringBuilder.insert(11, "HTML and ");` → **Welcome to HTML and Java**
`stringBuilder.delete(8, 11)` changes the builder to **Welcome Java**.
`stringBuilder.deleteCharAt(8)` changes the builder to **Welcome o Java**.
`stringBuilder.reverse()` changes the builder to **avaJ ot emocleW**.
`stringBuilder.replace(11, 15, "HTML")`
 changes the builder to **Welcome to HTML**.
`stringBuilder.setCharAt(0, 'w')` sets the builder to **welcome to Java**.

The toString, capacity, length, setLength, and charAt Methods in StringBuilder

- `+toString(): String` Returns a string object from the string builder.
- `+capacity(): int` Returns the capacity of this string builder.
- `+charAt(index: int): char` Returns the character at the specified index.
- `+length(): int` Returns the number of characters in this builder.
- `+setLength(newLength: int): void` Sets a new length in this builder.
- `+substring(startIndex: int): String` Returns a substring starting at startIndex.
- `+substring(startIndex: int, endIndex: int): String`
Returns a substring from startIndex to endIndex-1.
- `+trimToSize(): void` Reduces the storage size used for the string builder.

Problem: Checking Palindromes Ignoring Non-alphanumeric Characters

Assume the letters are not case-sensitive.

```
H:\work\JavaProg\2018Spring\notes08>javac PalindromeIgnoreNonAlphanumeric.java

H:\work\JavaProg\2018Spring\notes08>java PalindromeIgnoreNonAlphanumeric
Enter a string: This just a tsuj sihT
Ignoring non-alphanumeric characters,
is This just a tsuj sihT a palindrome? true

H:\work\JavaProg\2018Spring\notes08>java PalindromeIgnoreNonAlphanumeric
Enter a string: HelloWorld 12321 dlrow olleH
Ignoring non-alphanumeric characters,
is HelloWorld 12321 dlrow olleH a palindrome? false

H:\work\JavaProg\2018Spring\notes08>java PalindromeIgnoreNonAlphanumeric
Enter a string: HelloWorld 123321 dlrow olleH
Ignoring non-alphanumeric characters,
is HelloWorld 123321 dlrow olleH a palindrome? true
```

```
1 // Ch10, Liang's Book
2 import java.util.Scanner;
3 public class PalindromeIgnoreNonAlphanumeric {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6         System.out.print( "Enter a string: " );
7         String s = input.nextLine();
8
9         System.out.println( "Ignoring non-alphanumeric characters, \nis "
10             + s + " a palindrome? " + isPalindrome(s)
11         );
12     }
13
14     public static boolean isPalindrome (String s) {
15         String s1 = filter( s );    // eliminating non-alphanumeric chars
16         return s1.equals( reverse( s1 ) );    // s1.equalsIgnoreCase(s2)
17     }
18
19     public static String filter (String s) {
20         StringBuilder sb = new StringBuilder();
21         for (int i = 0; i < s.length(); i++)
22             if (Character.isLetterOrDigit( s.charAt(i) ))
23                 sb.append( s.charAt(i) );
24         return sb.toString();
25     }
26
27     public static String reverse (String s) {
28         return new StringBuilder(s).reverse().toString();
29     }
30 }
```

Regular Expressions (正则表达式)

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. Regular expression is a powerful tool for string manipulations.

You can use regular expressions for matching, replacing, and splitting strings.

Matching Strings

```
"Java".matches("Java");
```

```
"Java".equals("Java");
```

```
"Java is fun".matches("Java.*")
```

```
"Java is cool".matches("Java.*")
```

```
"Java is powerful".matches("Java.*")
```

Regular Expression Syntax

<i>Regular Expression</i>	<i>Matches</i>	<i>Example</i>
<code>x</code>	a specified character <code>x</code>	<code>Java</code> matches <code>Java</code>
<code>.</code>	any single character	<code>Java</code> matches <code>J..a</code>
<code>(ab cd)</code>	ab or cd	<code>ten</code> matches <code>t(en im)</code>
<code>[abc]</code>	a, b, or c	<code>Java</code> matches <code>Ja[uvw]a</code>
<code>[^abc]</code>	any character except a, b, or c	<code>Java</code> matches <code>Ja[^ars]a</code>
<code>[a-z]</code>	a through z	<code>Java</code> matches <code>[A-M]av[a-d]</code>
<code>[^a-z]</code>	any character except a through z	<code>Java</code> matches <code>Jav[^b-d]</code>
<code>[a-e[m-p]]</code>	a through e or m through p	<code>Java</code> matches <code>[A-G[I-M]]av[a-d]</code>
<code>[a-e&&[c-p]]</code>	intersection of a-e with c-p	<code>Java</code> matches <code>[A-P&&[I-M]]av[a-d]</code>
<code>\d</code>	a digit, same as <code>[0-9]</code>	<code>Java2</code> matches <code>"Java[\d]"</code>
<code>\D</code>	a non-digit	<code>\$Java</code> matches <code>"[\D][\D]ava"</code>
<code>\w</code>	a word character	<code>Java1</code> matches <code>"[\w]ava[\w]"</code>
<code>\W</code>	a non-word character	<code>\$Java</code> matches <code>"[\W][\w]ava"</code>
<code>\s</code>	a whitespace character	<code>"Java 2"</code> matches <code>"Java\s2"</code>
<code>\S</code>	a non-whitespace char	<code>Java</code> matches <code>"[\S]ava"</code>
<code>p*</code>	zero or more occurrences of pattern <code>p</code>	<code>aaaabb</code> matches <code>"a*bb"</code> <code>ababab</code> matches <code>"(ab)*"</code>
<code>p+</code>	one or more occurrences of pattern <code>p</code>	<code>a</code> matches <code>"a+b*"</code> <code>able</code> matches <code>"(ab)+.*"</code>
<code>p?</code>	zero or one occurrence of pattern <code>p</code>	<code>Java</code> matches <code>"J?Java"</code> <code>Java</code> matches <code>"J?ava"</code>
<code>p{n}</code>	exactly <code>n</code> occurrences of pattern <code>p</code>	<code>Java</code> matches <code>"Ja{1}.*"</code> <code>Java</code> does not match <code>".{2}"</code>
<code>p{n,}</code>	at least <code>n</code> occurrences of pattern <code>p</code>	<code>aaaa</code> matches <code>"a{1,}"</code> <code>a</code> does not match <code>"a{2,}"</code>
<code>p{n,m}</code>	between <code>n</code> and <code>m</code> occurrences (inclusive)	<code>aaaa</code> matches <code>"a{1,9}"</code> <code>abb</code> does not match <code>"a{2,9}bb"</code>

Replacing and Splitting Strings

`+matches(regex: String): boolean`

Returns true if this string matches the pattern.

`+replaceAll(regex: String, replacement: String): String`

Returns a new string that replaces all matching substrings with the replacement.

`+replaceFirst(regex: String, replacement: String): String`

Returns a new string that replaces the first matching substring with the replacement.

`+split(regex: String): String[]`

Returns an array of strings consisting of the substrings split by the matches.

Examples

```
String s = "Java Java Java".replaceAll("v\\w", "wi") ;  
Jawia Jawia Jawia
```

```
String s = "Java Java Java".replaceFirst("v\\w", "wi") ;  
Jawia Java Java
```

```
String[] s = "Java1HTML2Perl".split("\\d");  
{ "Java", "HTML", "Perl" }
```