Class java.lang.Object

Class java.lang.Object is a common superclass of all classes.
All objects, including arrays, inherit methods of this class:

- equals(Object obj) – tests if two objects are equal
- clone() – creates a copy of this object
- toString() – returns a string representation of the object
- hashCode() – returns a hash code value for the object
- getClass() – returns information about the class of the object
- finalize() – called by the garbage collector before the memory is freed
- wait(), notify(), notifyAll() – for synchronization of threads

References

There are three reference types in Java:

- class references
- interface references
- array references

There are two types of objects in Java:

- class instances
- arrays

References are **pointers** to objects. They can have **null** value and there can be many references to the same object.

The class `java.lang.Object` is a superclass of all other class. A variable of type `Object` can hold a reference to an instance of a class or to an array.

```
int[] a = { 3, 1, 5 };
Object o = a;
Object p = new Object[10];
```

References (cont.)

It is possible to compare two references using operators == and !=:

- The result of == is **true** if both references point to the same object or if they are both **null**.
- The result of == is **false** otherwise.
- The operator != works as a negation of ==.
- A compile-time error occurs if it is impossible to convert the type of either operand to the type of the other by a casting conversion.

```
Point a = new Point(10, 20);
Point b = new Point(10, 20);
System.out.println(a == b);   // prints 'false'
Object c = a;
System.out.println(a == c);   // prints 'true'
```

Method `equals()`

The method

```
public boolean equals(Object obj)
```

defined in the class `java.lang.Object` can be used to compare two different object if they are the same.
This method can be overridden in subclasses. If it is not overridden it behaves as if the test

```
(this == obj)
```

was used.

The method `equals()` implements an equivalence relation:

- **reflexive** – `x.equals(x)` should return **`true`**,
- **symmetric** – if `x.equals(y)` then also `y.equals(x)`,
- **transitive** – if `x.equals(y)` and `y.equals(z)` then also `x.equals(z)`,
- **consistent** – `x.equals(y)` should return always the same value, if objects pointed to by `x` and `y` has not changed,
- `x.equals(`**`null`**`)` should return **`false`**.

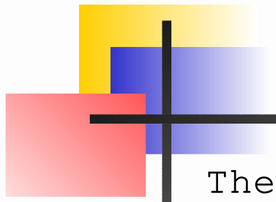
Method `equals()` (cont.)

```
class Point {
    private int x, y;

    public boolean equals(Object obj) {
        if (!(obj instanceof Point)) return false;
        Point p = (Point)obj;
        return (x == p.x && y == p.y);
    }

    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
}
```

```
Point a = new Point(10, 20);
Point b = new Point(10, 20);
Point c = new Point(30, 20);
System.out.println(a == b);        // prints 'false'
System.out.println(a.equals(b));   // prints 'true'
System.out.println(a.equals(c));   // prints 'false'
```



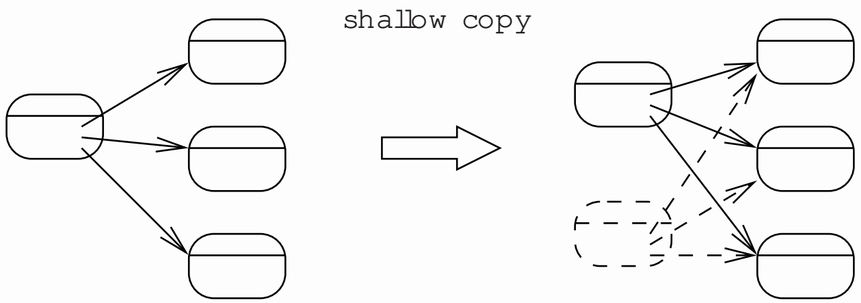
Copying Objects

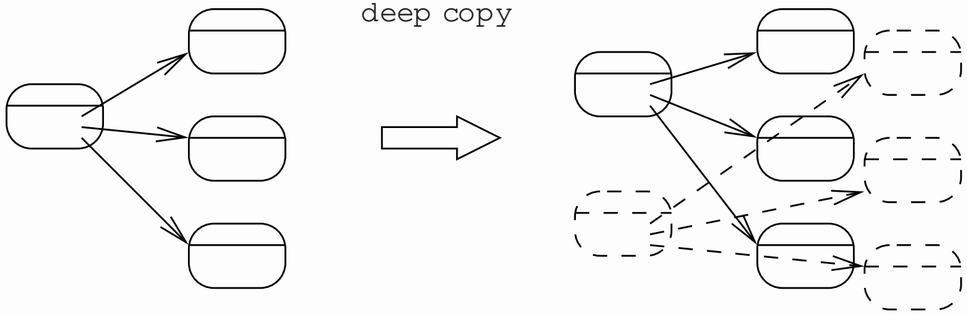
The clone() method is intended for creation of a copy of an object. The simplest way to make your class cloneable, is to add **implements** Cloneable to class's declaration. For some classes the default behavior of Object's clone() method works just fine. Other classes need to override clone to get correct behavior.

- **Shallow Copy**
  A clone of an original object is created only. All instance variables of the clone have the same values as the ones of the original, i.e. if a variable holds reference to an object, the original and the copy refer to the same object.
- **Deep Copy**
  Copies of an original object and all its instance variables are created. Then, any modification of the original does not affect its copy and vice versa.

Note: The clone() should never use **new** to create the clone and should not call constructors. Instead, the method should call **super**.clone(), which creates an object of the correct type and allows the hierarchy of superclasses to perform the copying necessary to get a proper clone.

Copying Objects (cont.)





Garbage Collection

When garbage collector is ready to release a memory used for an object, it will first call `finalize()` method, and only then the memory is reclaimed. Usage of `finalize()` gives the ability to perform some important cleanup **at the time of garbage collection.**

```
protected void finalize()
```

If there is some activity that must be performed before an objects is no longer need, the activity must be performed by programmer. Java has no destructor or similar concept, so an ordinary method performing this cleanup must be created.

Note: It is not good idea to rely on `finalize()` being called, and separate cleanup functions should be created and called explicitly.

Garbage collection can be characterized as follows:

- garbage collection is not destruction,
- some objects might not get garbage-collected,
- garbage collection is only about memory.

Strings

Strings are sequences of characters (primitive type **char**).

There are two classes in Java that can be used to represent strings (both are from the package `java.lang`):

- **String**
- **StringBuffer**

All string literals, such as `"abc"`, are represented as instances of the class **String**.

Strings are constants, their values cannot be changed after they are created.

The class **StringBuffer** supports mutable strings.

Strings in Java are not arrays of characters. This means that **char**`[]` is not `String` and vice versa.

However, character arrays can be used when we work with strings. Also both classes **String** and **StringBuffer** use character arrays in their internal implementation.

Class `String`

The easiest way how strings can be created is to use string literals:

```
String s = "abc";
```

Notice that no operator **new** is used in this case. The object of the class `String` is created automatically in this case.

The class `String` has many different constructors:

- `String()`
- `String(String original)`
- `String(StringBuffer buffer)`
- `String(char[] value)`
- `String(char[] value, int offset, int count)`
- `String(byte[] bytes, String charsetName)`
- `...`

Class `String` (cont.)

```
char data[] = {'a', 'b', 'c'};
String s = new String(data);

char data2[] = {'a', 'b', 'c', 'd', 'e', 'f'};
String t = new String(data2, 2, 3);  //  t = "cde";
```

The most important methods in the class `String`:

- **int** `length()` – returns the length of the string
- **char** `charAt(int index)` – returns the character at the specified index
- **boolean** `equals(Object obj)` – compares two strings

```
String s = "abcdef";
System.out.println(s.length());          // prints '6'
System.out.println(s.charAt(5));         // prints 'f'
System.out.println(s.equals("abcdef")); // prints 'true'
System.out.println(s.equals("hello"));  // prints 'false'
```

Class `String` (cont.)

Methods that transform strings to arrays:

- `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
- `char[] toCharArray()`
- `byte[] getBytes(String charsetName)`
- `byte[] getBytes()`

Methods for comparison of strings:

- `boolean equals(Object obj)`
- `boolean equalsIgnoreCase(String str)`
- `int compareTo(String str)`
- `int compareToIgnoreCase(String str)`
- `boolean contentEquals(StringBuffer sb)`

Note: Methods `compareTo()` and `compareToIgnoreCase()` return a negative integer, zero, a positive integer if the the specified string is greater than, equal to, or less than this string.

Class `String` (cont.)

Methods for searching for a character in a string:

- `int indexOf(int ch)`
- `int indexOf(int ch, int fromIndex)`
- `int lastIndexOf(int ch)`
- `int lastIndexOf(int ch, int fromIndex)`

Methods for searching for a substring in a string:

- `int indexOf(String str)`
- `int indexOf(String str, int fromIndex)`
- `int lastIndexOf(String str)`
- `int lastIndexOf(String str, int fromIndex)`

Note: The value `-1` is returned if the searched character or substring is not found.

Class `String` (cont.)

The methods for obtaining substrings:

- `String substring(int beginIndex)`
- `String substring(int beginIndex, int endIndex)`

```
String s = "abcdef";
System.out.println(s.substring(2,5)); // prints 'cde'
```

The methods for comparison of substrings:

- `boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)`
- `boolean startsWith(String prefix, int offset)`
- `boolean startsWith(String prefix)`
- `boolean endsWith(String suffix)`

Class `String` (cont.)

Other methods that manipulate strings:

- String toLowerCase()
- String toUpperCase()
- String concat(String str) – concatenates the specified string to the end of this string
- String replace(char oldChar, char newChar) – replaces all occurrences of oldChar with newChar
- String trim() – removes leading and trailing whitespace

There are also static methods called `valueOf()` that transform different types of values to strings:

- static String valueOf(boolean b)
- static String valueOf(char c)
- static String valueOf(int i)
- ...

Strings - Example

```
public class Filename {
    private String fullPath;
    private char pathSeparator, extensionSeparator;

    public Filename(String str, char sep, char ext) {
        fullPath = str;
        pathSeparator = sep;
        extensionSeparator = ext;
    }

    public String getExtension() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        return fullPath.substring(dot + 1);
    }

    public String getFilename() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(sep + 1, dot);
    }
```

Strings - Example

```
    public String getPath() {
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(0, sep);
    }
}
```

The following code illustrates usage of Filename:

```
Filename myHomePage = new Filename("/home/mem/index.html",
                                   '/', '.');
System.out.println("Extension = " + myHomePage.getExtension());
System.out.println("Filename = " + myHomePage.getFilename());
System.out.println("Path = " + myHomePage.getPath());
```

Produced output:

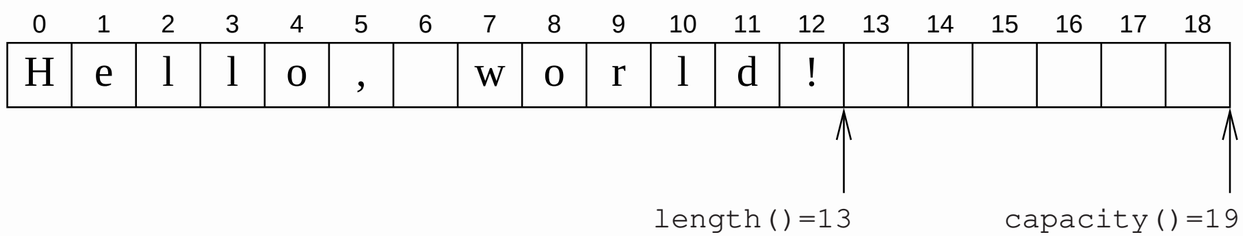
```
Extension = html
Filename = index
Path = /home/mem
```

Class StringBuffer

The class StringBuffer implements a mutable sequence of characters, the length and content of the sequence can be changed through certain method calls.

The most important methods:

- int length()
- void setLength(int newLength)
- char charAt(int index)
- void setCharAt(int index, char ch)
- int capacity()
- void ensureCapacity(int minimumCapacity)



Class `StringBuffer` (cont.)

The constructors:

- StringBuffer()
- StringBuffer(int length)
- StringBuffer(String str)

The most important methods used to modify a string buffer are:

- **append** – adds characters at the end of the buffer,
- **insert** – adds characters at a specified point.

```
StringBuffer b = new StringBuffer("abcd");
b.append("ef");       // 'b' contains 'abcdef'
b.insert(3, "ghi");   // 'b' contains 'abcghidef'
```

The contents of a string buffer can be transformed into a string using the `toString()` method:

```
String s = b.toString();   // 's' contains 'abcghidef'
```

Class `StringBuffer` (cont.)

Methods append and insert are overloaded so as to accept data of any type:

- StringBuffer append(String str)
- StringBuffer append(StringBuffer sb)
- StringBuffer append(Object obj)
- StringBuffer append(char[] str)
- StringBuffer append(boolean b)
- StringBuffer append(char c)
- StringBuffer append(int i)
- StringBuffer append(double d)
- ...
- StringBuffer insert(int offset, String str)
- StringBuffer insert(int offset, char[] str)
- ...

Class `StringBuffer` (cont.)

It is also possible to delete characters:

- StringBuffer delete(int start, int end)
- StringBuffer deleteCharAt(int index)

Other methods that manipulate string buffers:

- StringBuffer replace(int start, int end, String str)
- String substring(int start)
- String substring(int start, int end)

The methods for searching in string buffers:

- int indexOf(String str)
- int indexOf(String str, int fromIndex)
- int lastIndexOf(String str)
- int lastIndexOf(String str, int fromIndex)

Operator +

Operator + can be used for concatenation of strings:

```
String a = "Hello";
String b = ", world";
String c = a + b;  // c = "Hello, world"
```

It is also possible to use +=:

```
String a = "Hello";
a += ", world";   ;  // a = "Hello, world"
```

If at least one of operands of + is a string, the other operand is transformed into a string automatically:

```
String a = "Result=" + (3 + 2) ;  // a = "Result=5"
String b = "Result=" + 3 + 2 ;    // b = "Result=32"
```

Operator + (cont.)

String buffers are used by the compiler to implement the string concatenation operator +. For example, the code:

```
String x;
x = "a" + 4 + "c";
```

is compiled to the equivalent of:

```
String x;
x = new StringBuffer().append("a").append(4).append("c")
                                  .toString();
```

It is usually more efficient to manipulate a StringBuffer then to manipulate String. A new instance of String is created after every operation.

StringBuffer - Example

```
class StringBufferDemo {
    static String toString(int[] a) {
        StringBuffer buf = new StringBuffer("{ ");
        for (int i = 0; i < a.length; i++) {
            if (i > 0) buf.append(", ");
            buf.append(a[i]);
        }
        buf.append(" }");
        return buf.toString();
    }

    public static void main(String[] args) {
        int[] a = { 81, 32, 53, 21, 82 };
        String s = toString(a);
        System.out.println(s);
    }
}
```

```
Output: { 81, 32, 53, 21, 82 }
```