

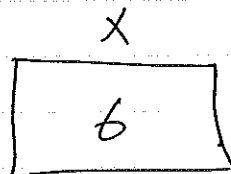
## - JAVA DATA TYPES -

JAVA HAS TWO KINDS OF DATA TYPES

- o PRIMITIVE TYPES (int, double, char, boolean)
- o REFERENCE TYPES (classes)

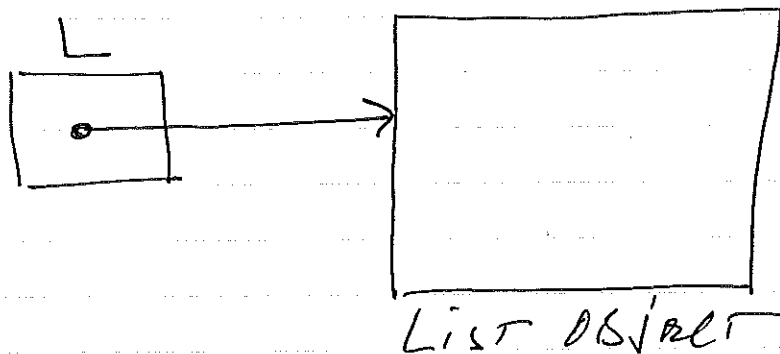
A VARIABLE OF PRIMITIVE TYPE IS A SYMBOLIC NAME FOR A MEMORY LOCATION WHICH STORES SOME DATA.

```
int x = 6;
```



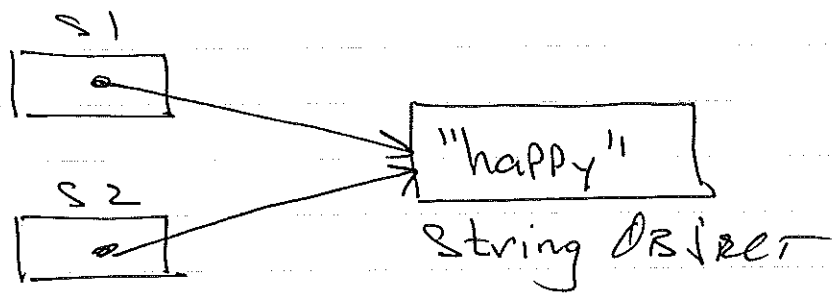
A VARIABLE OF REFERENCE TYPE IS A SYMBOLIC NAME FOR A MEMORY LOCATION WHICH STORES AN ADDRESS OF SOME OBJECT

```
IntegerList L = new IntegerList();
```



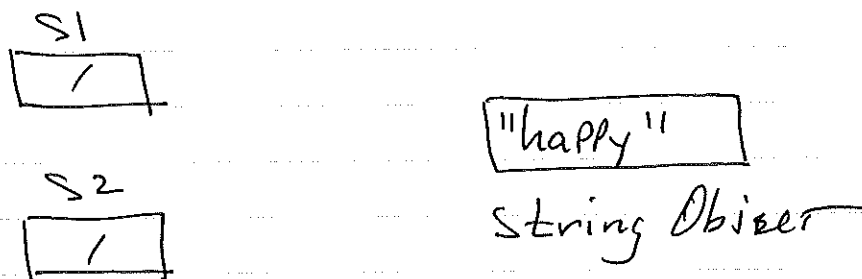
(IN C WE CALL THESE POINTER TYPES)

```
String s1 = new String("happy");
String s2 = s1;
```



CHANGE TO THE OBJECT REFERENCED BY S1 ARE ALSO SEEN THROUGH S2.

```
s1 = s2 = null;
```



JAVA HAS A GARBAGE COLLECTION MECHANISM WHICH MARKS FOR DELETION ALL OBJECTS WHICH NO LONGER ARE REFERENCED IN A PROGRAM. THAT MEMORY IS PERIODICALLY DE ALLOCATED AND MADE AVAILABLE FOR FUTURE USE.

```
//  
// IntegerList.java  
( // Array based implementation of Integer List ADT  
//  
public class IntegerList{  
  
    public static final int MAX_LENGTH = 1000;  
    private int[] item;          // array of IntegerList items  
    private int numItems;       // number of items in this IntegerList  
  
    // arrayIndex  
    // transforms a List index to an Array index  
    private int arrayIndex(int listIndex){  
        return listIndex-1;  
    }  
  
    // IntegerList  
    // default constructor for the IntegerList class  
    public IntegerList(){  
        item = new int[MAX_LENGTH];  
        numItems = 0;  
    }  
  
    // isEmpty  
    // pre: none  
    // post: returns true if this IntegerList is empty, false otherwise  
    public boolean isEmpty(){  
        return(numItems == 0);  
    }  
  
    // size  
    // pre: none  
    // post: returns the number of elements in this IntegerList  
    public int size() {  
        return numItems;  
    }  
  
    // get  
    // pre: 1 <= index <= size()  
    // post: returns item at position index  
    public int get(int index){  
        if( index<1 || index>numItems ){  
            System.err.println("List Error: get() called with index out of bounds");  
            System.exit(1);  
        }  
        return item[arrayIndex(index)];  
    }  
}
```

```
( ) // add
// inserts newItem in this IntegerList at position index
// pre: size() < MAX_LENGTH, 1 <= index <= size()+1
// post: !isEmpty(), items to the right of newItem are renumbered
public void add(int index, int newItem){
    if( numItems == MAX_LENGTH ){
        System.err.println("List Error: add(,) called on full List");
        System.exit(1);
    }
    if( index<1 || index>(numItems+1) ){
        System.err.println("List Error: add(,) called with index out of bounds");
        System.exit(1);
    }
    for(int i=numItems; i>=index; i--) item[arrayIndex(i+1)] = item[arrayIndex(i)];
    item[arrayIndex(index)] = newItem;
    numItems++;
}
```

```
// remove
// deletes item from position index
// pre: 1 <= index <= size()
// post: items to the right of deleted item are renumbered
public void remove(int index){
    if( index<1 || index>numItems ){
        System.err.println("List Error: remove() called with index out of bounds");
        System.exit(1);
    }
    for(int i=index+1; i<=numItems; i++){
        item[arrayIndex(i-1)] = item[arrayIndex(i)];
    }
    numItems--;
}
( )
```

```
// removeAll
// pre: none
// post: isEmpty()
public void removeAll(){
    item = new int[MAX_LENGTH];
    numItems = 0;
}
}
```

## - INHERITANCE -

WE CAN CREATE A SUBCLASS OF AN EXISTING CLASS USING THE KEYWORD extends

```
class child_class extends parent_class {
    :
}
```

THE PARENT CLASS IS ALSO CALLED THE BASE CLASS OR SUPER CLASS. THE CHILD CLASS IS ALSO CALLED DERIVED CLASS OR SUBCLASS.

NOTE: SUBCLASS & SUPERCLASS CAN BE MISLEADING.

ALL MEMBERS (VARIABLES & METHODS) BELONGING TO THE PARENT ALSO BELONG TO THE CHILD. THE CHILD CLASS CAN ALSO CONTAIN MORE MEMBERS, WHICH IT DEFINES.

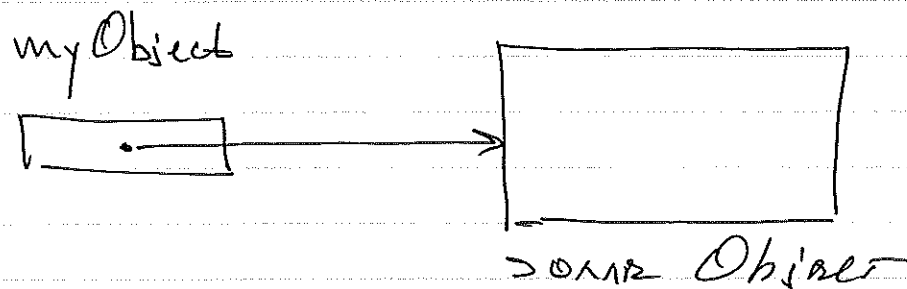
ALL CLASSES IN JAVA ARE SUBCLASSES OF THE Object class.

A CHILD CLASS CAN OVERLOAD OR OVERRIDE ONE OF ITS PARENT'S METHODS.

- OVERLOAD: NEW METHOD WITH SAME NAME  
DISTINCT SIGNATURE
- OVERRIDE: REDEFINE PARENT METHOD,  
MUST HAVE SAME SIGNATURE.

For instance, `Object` contains a public method `toString()` which returns a string representation of a class instance.

By default, `toString()` returns the address of an object in the VM.



```
System.out.println(myObject.toString());
```

Prints the address. Actually `println` calls `toString()` implicitly:

```
System.out.println(myObject)
```

TO override this method do e.g.

```
public String toString() {
```

```
!
```

```
}
```

in the child class

EX

```

public String toString() {
    String s = "";
    for (int i=0; i < numItems; i++) {
        s = s + String.valueOf(item[i]) + " ";
    }
    return s;
}

```

Object also contains a public method called equals(), which returns true iff two arguments are the same object, i.e. compares addresses in v.m.

OFTEN WE WISH TO OVERRIDE equals().

EX.

```

public boolean equals(Object rhs) {
    if (rhs instanceof IntegerList) {
        boolean equal = (numItems == rhs.numItems);
        int i = 0;
        while (equal && i < numItems) {
            equal = (item[i] == rhs.item[i]);
        }
        return equal;
    }
    return false;
}

```

CAST AS IntegerList

CAST AS IntegerList

NOTICE WE DECLARE equals TO TAKE AN OBJECT (NOT IntegerList) THEN CHECK THAT ARGUMENT IS AN INSTANCE OF IntegerList.

IF WE DID

```

public boolean equals (IntegerList rhs) {
    // NO CASTS
}
    
```

WE WOULD BE OVERLOADING equals, NOT OVERRIDING IT.

JAVA DISTINGUISHES BETWEEN DIFFERENT METHODS WITH THE SAME NAME BY THEIR PARAMETER LIST AND RETURN VALUE (i.e. THEIR SIGNATURE). USE THE SAME SIGNATURE TO OVERRIDE, DIFFERENT SIGNATURES TO OVERLOAD.

WHY NOT JUST OVERLOAD (AS ABOVE) INSTEAD OF OVERRIDE FOR IntegerList? IF ALL WE'RE GOING TO DO IS COMPARE TWO IntegerList like

```

A.equals(B);
    
```

THIS IS FINE, OFTEN THOUGH ADTs EXPORT SERVICE TO OTHER ADTs, equals in ONE CLASS WILL THEREFORE CALL equals IN ANOTHER CLASS.

e.g. IntegerList could be a subclass of some other class (BOTH ADT.) In such a case equals will call the equals() method belonging to the superclass Object, which must therefore be overridden.

EXERCISE: DO BOTH

```
public boolean equals(Object rhs) {
    System.out.println("in Object's equals");
}
```

```
public boolean equals(IntegerList rhs) {
    System.out.println("in IntegerList's equals");
}
```

Then write a client for IntegerList:

```
public class IntegerListClient {
    public static void main(String[] args) {
        // Create IntegerLists A and B
        A.equals(B); // try A.equals(Object B);
    }
}
```

To see which equals is called. } And. IntegerList's

Exercise :

(1) ADD methods toString() & equals() TO class IntegerList which OVERRIDE THE CORRESPONDING METHODS IN class Object

(2) Write A program called IntegerListTest.java THAT EXERCISES ALL METHODS IN IntegerList .

```
// IntegerListTest.java
```

```
class IntegerListTest {
```

```
    public static void main(String[] args) {
```

```
        // do stuff with IntegerList
```

```
    }
```

```
}
```