Review

- Variables whose type is a class are references
- Alias of a variable is a different variable which refers to the same object
- Aliases can be used to represent shared references/shared information
- Question: When are two references (of the same type) equivalent?
- Reference equality
  - Pointer equality
  - Structural equality
    - Shallow
    - Deep

Review

- Two references are *pointer-equal* if they are aliases (i.e. they refer to the same object)
- Two references are *structurally-equal* if the attributes of the objects they refer to are equal
  - Two references are *shallowly-equal* if the attributes of the objects they refer to are pointer-equal
  - Two references are *deeply-equal* if the attributes of the objects they refer to are structurally-equal
- Copying or cloning: creating a *different* (i.e. not pointer-equal) object which is structurally-equal to the original
  - Copy $\left\{ \begin{array}{l} \text{Shallow} \\ \text{Deep} \end{array} \right.$
  - A copy is shallow if it is shallowly-equal to the original
  - A copy is deep if it is deeply-equal to the original

Passing parameters by reference

```
class A {
  int x;
}
class B {
  static void m(A u)
  {
    u.x++;
  }
  static void t(int x)
  {
    x++;
  }
}
```

Passing parameters by reference

```
class Test {
  void p()
  {
    A q = new A();
    q.x = 3;
    B.m(q);
    System.out.println(q.x);
    B.t(q.x);
    System.out.println(q.x);
  }
}
```

Garbage collection

- Each use of the keyword new creates an object that takes up space in memory:

```
for (int i=0; i<10000; i++) {
  A some_a = new A();
  some_a.do_something();
}
```

- But there is a limited amount of memory
- The JVM uses Garbage Collection as a mechanism to manage memory
- Garbage collection consists of freeing up any unused memory
- The Garbage collector keeps track of the number of references (aliases) to each object. When an object has no references to it, its memory is reclaimed and it can be recycled.

Garbage collection (contd.)

- When can an object have no more references to it?
- When reassigning a reference:

```
A x = new A();
x = new A();
x = null;
```

- When a method returns (and the local reference is not passed to another object.)

```
void do_something()
{
  A temp = new A();
  temp.p();
  // ...
}
```

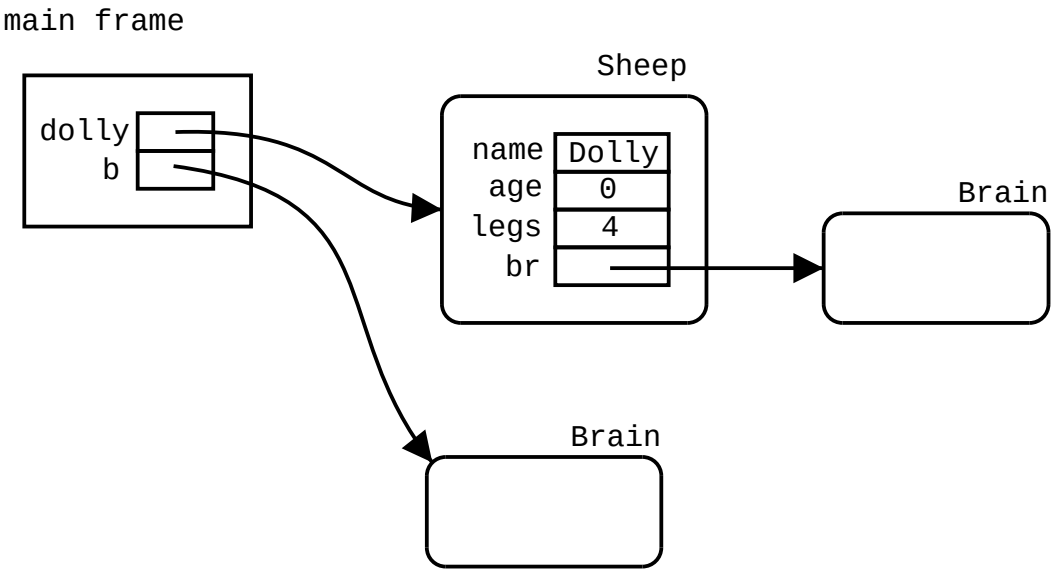
Example

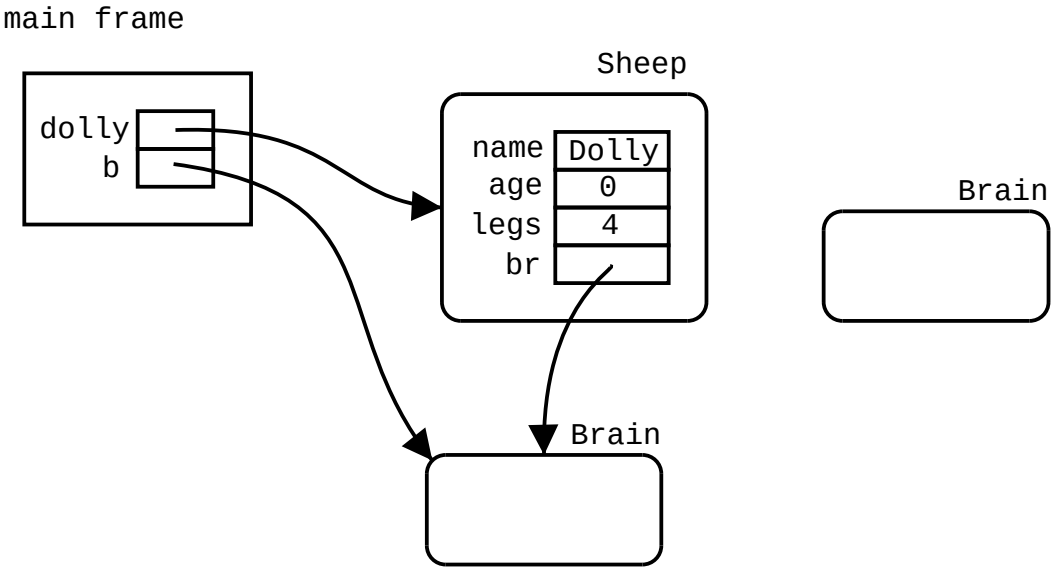
```
class Brain {
  //...
}
class Sheep {
  String name;
  int age, legs;
  Brain br;
  Sheep(String n)
  {
    name = n;
    age = 0;
    legs = 4;
    br = new Brain();
  }
  void brain_transplant(Brain new_brain)
  {
    br = new_brain;
  }
}
```

Example (contd.)

```
public class BrainTest {
  public static void main(String[] args)
  {
    Sheep dolly = new Sheep(“Dolly”);
    Brain b = new Brain();
    dolly.brain_transplant(b);
  }
}
```

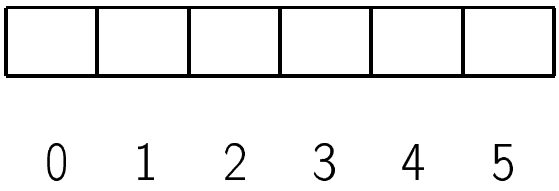
Example (contd.)





Arrays

- An *array* is an indexed sequence of variables of the same type. By indexed we mean that the variables are consecutive in memory and each of them has an index, with 0 being the first, 1 the second, and so on.



- Each variable in the array is called a *position*, a *cell* or a *slot*, and as any variable, it can contain a value.

- Arrays are declared as follows:

  type `[]` *name* `;`

- Where *type* is any data type (primitive or user-defined).

Arrays (contd.)

- For example an array of integers called `mylist` which is declared as

```
int[] mylist;
```

- In an array declaration *type[]* is the type of the array, and *type* is its *base type*. (An array of integers is not the same as a single integer.)

- Arrays can have as base type a class.

- For example, if we have a class Mouse then an array of mice is declared as:

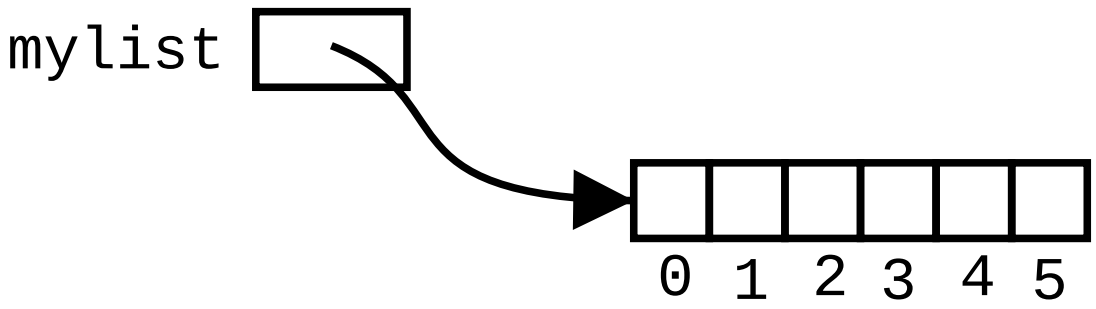
```
Mouse[] mouse_list;
```

Arrays (contd.)

- But declaring an array does not create the array itself, only a reference.
- To create an array we use the new keyword.

```
mylist = new int[6];
```

- Where the variable mylist is actually a reference to the aray itself



Array access

- To access individual elements of an array we use the indexing operator [.]: If variable is a reference to an array, and *number* is a positive integer, or 0, then the position *number* can be accessed by

  variable[*number*]

- For example `mylist[0]` refers to the first position of mylist, `mylist[1]` to the second, `mylist[2]` to the third, and so on.

- To write a value in the array, we can use the assignment operator:

  variable[*number*] = *expression*;

- Where *expression* must be of the same type as the base type of the array.

Processing arrays

- Processing arrays is a generalization of processing strings.
- `a[i]` is analogous to `s.charAt(i)`, but only for reading the `i`-th, not for writing: `charAt` cannot be used for modifying a string. This is: `s.charAt(i) = expr;` is illegal syntax.
- Use loops to traverse an array.
- The length of an array `a` can be obtained by the expression `a.length`
- This is independent of the number of slots that hold a value

Example 1

- Finding the minimum number in an array

```
static double find_min(double[] a)
{
  int index;
  double minimum;
  index = 0;
  minimum = 999999999.9;
  while (index < a.length) {
    if (a[index] < minimum) {
      minimum = a[index];
    }
    index++;
  }
  return minimum;
}
```

Example 2

- Returning the index where the minimum is located

```
static int find_min(double[] a)
{
  int index, min_index;
  double minimum;
  index = 0;
  min_index = 0;
  minimum = a[0];
  while (index < a.length) {
    if (a[index] < minimum) {
      minimum = a[index];
      min_index = index;
    }
    index++;
  }
  return min_index;
}
```