# Static variables

- The attributes of a class are normal variables.

- The values of these attributes are individual to each object in a class.

```
public class A {
  int x;
}
public class B {
  void m()
  {
    A u = new A();
    A v = new A();
    u.x = 5;
    v.x = -7;
    // Here, u.x == 5 and v.x == -7
  }
}
```

McGill

# Static variables (contd.)

- Static variables are attributes of the class, not of the objects
- Static variables are shared between all the objects in a class

```
public class A {
  static int x;
}
public class B {
  void m()
  {
    A u = new A();
    A v = new A();
    u.x = 5;
    v.x = -7;
    // Here, u.x == -7 and v.x == -7
  }
}
```

# Example

```
public class A
{
    void p()
    {
        System.out.println(‘‘Hello’’);
    }
    static void q()
    {
        System.out.println(‘‘Good bye’’);
    }
}
```

(Note: Classes can have both static and non-static methods)

# Calling static methods

- A call to a static method takes the form

  *class_name.method(arg1,arg2,...,argn)*

- When the method is called, the corresponding frame does not have a reference to `this`, because there is no object receiving the message.

- It can also take the form

  *object_reference.method(arg1,arg2,...,argn)*

- But the object will be ignored

# Static methods access

- Since the frame of a static method does not have a reference to an object, static methods cannot access attributes of an object

```
public class A
{
  int n;
  void p()
  {
    System.out.println(n); //OK
  }
  static void q()
  {
    System.out.println(n); //WRONG
  }
}
```

# Accessing static methods from non-static methods

```
public class A
{
    void p()
    {
        System.out.println(``Hello'');
        q();
    }
    static void q()
    {
        System.out.println(``Good bye'');
    }
}
```

... is OK

# Accessing non-static methods from static methods

```
public class A
{
    void p()
    {
        System.out.println(``Hello'');
    }
    static void q()
    {
        System.out.println(``Good bye'');
        p();
    }
}
```

... is **not** OK, because in method q, there is no reference "`this`" to an object to which the message "`p()`" would be sent.

# When to use each kind of method

- Non-static methods are used to describe the behaviour of objects.
- Static methods are used to describe functions, or services that a class provides, independently of any object of that class.

# Separation of concerns

- Separate the *User Interface* from the *Logic* of the program in separate modules
  - User Interface: Anything that specifies interaction with the user (i.e. print, keyboard operations.)
  - Logic:
    * Data representation (meaningful classes)
    * Algorithms (methods that solve the problem(s))

- Saparating the user interface from the logic increases the maintainability

# Separation of concerns

```
public class BankAccount {
    private float balance;
    public BankAccount() { balance = 0.0f; }
    public void deposit(float amount)
    {
        balance += amount;
    }
    public void withdraw(float amount)
    {
        if (amount <= balance)
            balance -= amount;
    }
    public float getBalance()
    {
        return balance;
    }
}
```

# Separation of concerns

```
public class Banking {
  static float enterAmount()
  {
    float a = 0.0f;
    do {
      System.out.print(``Enter an amount: '');
      a = Keyboard.readFloat();
    } while (a < 0.0f);
    return a;
  }
  static void printBalance(BankAccount a)
  {
    System.out.print(``Balance = '');
    System.out.println(a.getBalance());
  }
```

McGill

```
  public static void main(String[] args)
  {
    BankAccount a = new BankAccount();
    float x;
    x = enterAmount();
    a.deposit(x);
    x = enterAmount();
    a.withdraw(x);
    printBalance(a);
  }
}
```

# Separation of concerns

```
public class BankAccount {
    private float balance;
    public BankAccount() { balance = 0.0f; }
    public void deposit()
    {
        float amount = Keyboard.readFloat();
        balance += amount;
    }
    public void withdraw()
    {
        float amount = Keyboard.readFloat();
        if (amount <= balance)
            balance -= amount;
    }
    public void getBalance()
    {
        System.out.println(balance);
    }
}
```

# Passing parameters

- Parameters are passed to a method in two different ways:
  - By value:
    * A copy of the argument is assigned to the parameter
    * Any changes to the parameter do not affect the caller's argument
    * Primitive values are passed by value
  - By reference
    * A reference to the argument is assigned to the parameter
    * Changes to the parameter may affect the caller's argument
    * Objects are passed by reference

# Passing parameters by value

```
class A {
  void f(int x)
  {
    x++;
  }
  void g()
  {
    int x = 3;
    f(x);
    System.out.println(x);
  }
}
```

# Passing parameters by reference

```
class B { int x; }
class A {
  void f(B u)
  {
    u.x++;
  }
  void g()
  {
    B u = new B();
    u.x = 3;
    f(u);
    System.out.println(u.x);
  }
}
```

# The null reference

- A variable whose type is a class is initialised to `null`.

- If a variable whose type is a class is not assigned an object (constructed with new,) and we try to access its attributes or methods, then a run-time error, called a "null-pointer exception" will occur.

- In the following example, if method `r` is called, a null pointer exception will occur:

```
class B { int x; }
class A {
  void f(B u)
  {
    u.x = 7;   // Null pointer exception
  }
  void g()
  {
    B v;  // v == null
    f(v);
  }
}
```

McGill

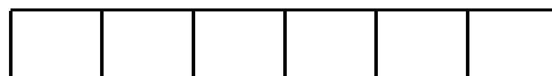
# The null reference (contd.)

- We can avoid these errors by using an explicit check for a valid reference:

```
class B { int x; }
class A {
  void f(B u)
  {
    if (u != null)
      u.x = 7;
  }
  void g()
  {
    B v;  // v == null
    p(v);
  }
}
```

# 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.



0 1 2 3 4 5

- 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:

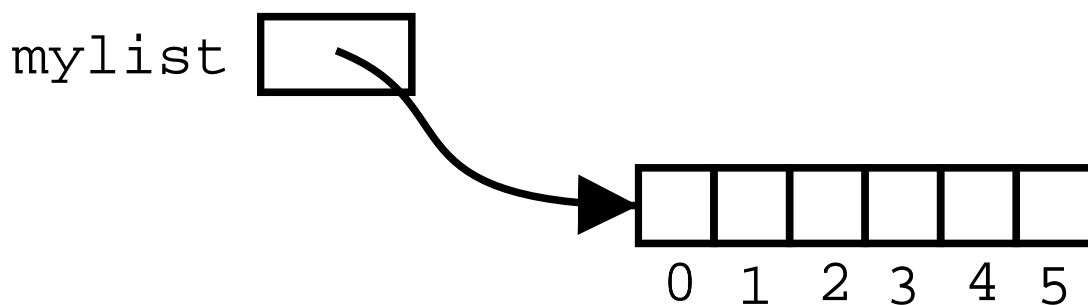
```
Mouse[] mouse_list;
```

# 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

- Filling an array

```
static void fill(double[] a)
{
  int index;
  index = 0;
  while (index < a.length) {
    a[index] = Math.random();
    index++;
  }
}
```

# Example 2

- 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 3

- 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;
}
```

# Processing arrays: safety

- Since arrays are references, it is often useful to check whether they are null or not before using them, to avoid null-pointer exceptions.
- If the array has as base type a class, it is also useful to check that each slot which will be processed or accessed is not null.
- For example:

```
class A { int x; }
class B {
  static void m(A[] list)
  {
    if (list != null) {
      for (int i = 0; i < list.length; i++) {
        if (list[i] != null) {
          list[i] = 2 * i;
        }
      }
    }
  }
```

# Initializing arrays

- If we have a class

```
class B {
  int n;
  B(int x) { n = x; }
}
```

- and somewhere else we declare and create an array

```
B[] list = new B[7];
```

- Then all the slots in the array will be initialized to `null`. This is, the constructor for B will not be called. If we want an object created in each slot, we have to do it explicitely:

```
for (int i=0; i < list.length; i++)
  list[i] = new B(3);
```

# Initializing arrays

- Arrays can be initialized with default values using the syntax:

  *type*`[]` *var* `= {` *expr1*`,` *expr2*`, ...,` *exprn* `};`

Where each *expri* is of type *type*.

- For example:

```
int[] a = { 1, 1, 2, 3, 5 };
Z[] u = { new Z(), new Z() };
```

# The end