Review

- Objects as first class values and Aggregation
- References and aliases
- Encapsulation
- Method overloading
- Static variables and methods

Objects as first class values

- Objects can be...
  - ...passed as parameters to a method
  - ...return as a result from a method
  - ...be attributes of other objects

- Aggregation:
  - When an attribute of an object is a reference to another object
  - The "has-a" relationship between objects

Objects as first class values

```java
public class Movie
{
  String title, director;
  Movie(String t, String d)
  {
    title = t;
    director = d;
  }
  void print()
  {
    System.out.println(title);
    System.out.println(director);
  }
}
```

McGill

Objects as first class values

- Objects can be passed as parameters to a method

```
public class Theater {
  void play(Movie m)
  {
    m.print();
  }
}

public class MovieApplication {
  public static void main(String[] args)
  {
    Movie m1;
    Theater t = new Theater();
    m1 = new Movie(“Les Invasions barbares”,
                    “Denys Arcand”);
    t.play(m1);
  }
}
```

McGill

Objects as first class values

- Objects can be attributes of other objects (Aggregation); Objects can be returned by methods

```
class MoviePair
{
  Movie m1, m2;

  void set_first(Movie m) { m1 = m; }
  void set_second(Movie m) { m2 = m; }

  Movie get_first()  { return m1; }
  Movie get_second() { return m2; }

  void play_both()
  {
    m1.play();
    m2.play();
  }
}
```

Objects as first class values

- Objects can be attributes of other objects

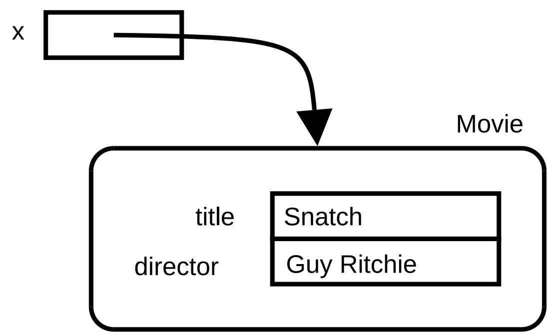
```
public class MovieApplication
{
  public static void main(String[] args)
  {
    Movie a, b;
    MoviePair pair;
    a = new Movie(``Lawrence of Arabia'', ``David Lea
    b = new Movie(``Snatch'', ``Guy Ritchie'');
    pair = new MoviePair();
    pair.set_first(a);
    pair.set_second(b);
    pair.play_both();
    Movie c = pair.get_second();
  }
}
```

References and Aliases

- Object references:

  A variable which is assigned an object, does not contain the object itself, but a *reference* to the object

```
Movie x = new Movie(“Snatch”, “Guy Ritchie”);
```

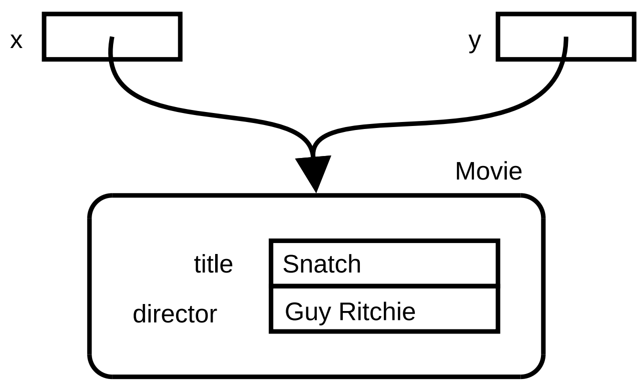


References and Aliases

- Aliases:

  Two variables are *aliases* if the refer to the same object

```
Movie x = new Movie("Snatch", "Guy Ritchie");
Movie y = x;
```



References and Aliases

- When the state of an object changes, the result affects all aliases.

```
class Movie
{
  String title, director;

  Movie(String t, String d) { ... }

  void change_title(String n)
  {
    title = n;
  }
  void print()
  {
    System.out.println(title);
    System.out.println(director);
  }
}
```

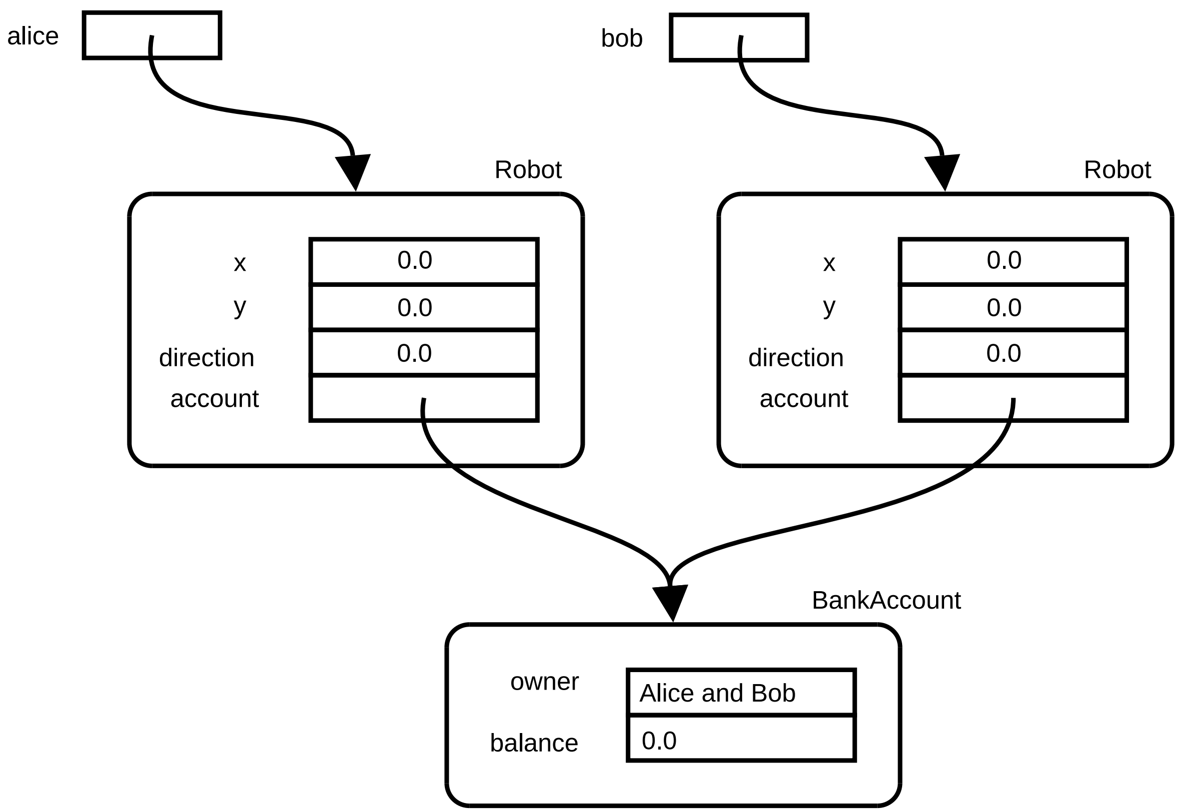
References and Aliases

- When the state of an object changes, the result affects all aliases.

```
Movie x = new Movie(“Snatch”, “Guy Ritchie”);
Movie y = x;
x.print();
y.change_title(“Pigs and diamonds”);
x.print();
```

References and Aliases

- Aliases can be used to represent shared information



References and Aliases

- Aliases can be used to represent shared information

```
class BankAccount
{
  double balance;
  String owner;

  BankAccount(String who) { ... }

  void withdraw(double amount) { ... }

  void deposit(double amount) { ... }

  double getBalance() { return balance; }
}
```

References and Aliases

- Aliases can be used to represent shared information

```
class Robot
{
  double x, y, direction;
  BankAccount account;

  Robot(double direction) { ... }

  void turn(double angle) { ... }

  void advance(double distance) { ... }

  void setAccount(BankAccount a)
  {
    account = a;
  }
  BankAccount getAccount() { return account; }
}
```

References and Aliases

- Aliases can be used to represent shared information

```
public class JointAccountsTest
{
  public static void main(String[] args)
  {
    Robot alice, bob;
    BankAccount account;

    alice = new Robot(0.0);
    bob = new Robot(0.0);
    account = new BankAccount(“Alice and Bob”);
    alice.setAccount(account);
    bob.setAccount(account);
  }
}
```

References and Aliases

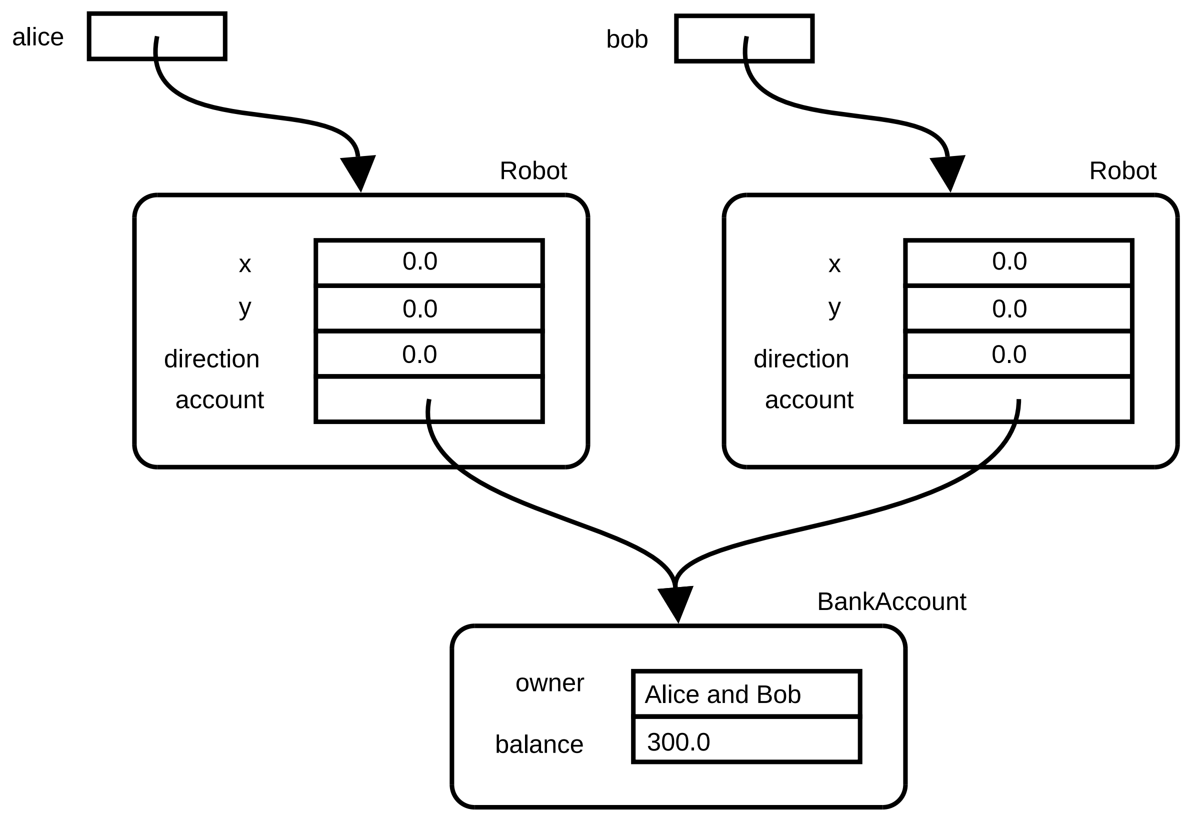
- When information is shared, changes to it affect all those who share it:

```
Robot alice, bob;
BankAccount account;

alice = new Robot(0.0);
bob = new Robot(0.0);
account = new BankAccount(“Alice and Bob”);
alice.setAccount(account);
bob.setAccount(account);

BankAccount a1 = alice.getAccount();
a1.deposit(300.0);
BankAccount a2 = bob.getAccount();
double bobs_money = a2.getBalance();
```

References and Aliases



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);
  }
}
```

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
    f(v);
  }
}
```

Encapsulation and visibility

- Abstraction and visibility
- Purpose of encapsulation:
  - Hiding the internal state of an object
  - Security: maintaining the integrity of data.
- Visibility modifiers (for attributes and methods): `public`, `private` and `protected`.

Encapsulation and visibility

- Attribute syntax:

  type identifier;

  or

  modifier type identifier;

  where *modifier* is one of:

  - private
  - public
  - protected

- For example:

```
private int age;
public String name;
protected long id;
```

- These modifiers are not applicable to local variables or parameters

Encapsulation and visibility

- Method syntax:

```
type methodname(parameters)
{
   // body
}
```

or

```
modifier type  methodname(parameters)
{
   // body
}
```

where *modifier* is one of:

- private
- public
- protected

Encapsulation and visibility

- Private: accessible only in its class
- Public: accessible everywhere
- Protected: accessible by anyone in the same "package"
- No modifier: the same as "protected"

Encapsulation to enforce integrity

```
public class BankAccount
{
  public double balance;
  public String owner;

  public BankAccount(String owner)
  {
    balance = 0.0;
    owner = who;
  }
  public void deposit(double amount)
  {
    balance = balance + amount;
  }
  public void widthdraw(double amount)
  {
    if (amount <= balance)
      balance = balance - amount;
  }
}
```

Encapsulation to enforce integrity

```
public class BankingApplication
{
  public static void main(String[] args)
  {
    BankAccount b;
    b = new BankAccount(“Zack”);
    b.deposit(500.0);
    b.balance = b.balance - 700.0;   // OK
  }
}
```

Encapsulation to enforce integrity

```
public class BankAccount
{
  private double balance;
  public String owner;

  public BankAccount(String owner)
  {
    balance = 0.0;
    owner = who;
  }
  public void deposit(double amount)
  {
    balance = balance + amount;
  }
  public void widthdraw(double amount)
  {
    if (amount <= balance)
      balance = balance - amount;
  }
}
```

Encapsulation to enforce integrity

```
public class BankingApplication
{
  public static void main(String[] args)
  {
    BankAccount b;
    b = new BankAccount(“Zack”);
    b.deposit(500.0);
    b.balance = b.balance - 700.0;    // ERROR
  }
}
```

Encapsulation to enforce integrity

```
public class BankingApplication
{
  public static void main(String[] args)
  {
    BankAccount b;
    b = new BankAccount("Zack");
    b.deposit(500.0);
    b.withdraw(700.0);    // OK
  }
}
```

Protected and packages

- Large Java programs are divided into *packages*
- A package is a collection of several classes
- A package is stored in a directory (folder)
- An attribute or method declared as protected can be accessed by any class in the same package

Privacy is relative

```
public class BankAccount
{
  private double balance;
  public String owner;

  public BankAccount(String owner) { ... }
  public void deposit(double amount) { ... }
  public void widthdraw(double amount)
  { ... }
  public void transfer(BankAccount other, double a
  {
    this.balance = this.balance - amount;
    other.balance = other.balance + amount;
  }
}
```

Privacy is relative

```
public class BankingApplication
{
  public static void main(String[] args)
  {
    BankAccount b1, b2;
    b1 = new BankAccount(“Zack”);
    b2 = new BankAccount(“Steph”);
    b1.deposit(500.0);
    b1.transfer(b2, 200.0);
  }
}
```

Method overloading

- In a given class there can be several methods with the same name...
- ...but the type or number of parameters must be different
- This is also true of constructors

Example

```
public class Dog
{
  void chaseTail()
  {
    System.out.println(“Woof! Woof!”);
  }
}
public class Cat
{
  void layDown()
  {
    System.out.println(“Meow”);
  }
}
```

Example

```
public class PetOwner
{
  void pet(Cat some_cat)
  {
    some_cat.layDown();
  }
  void pet(Dog some_dog)
  {
    some_dog.chaseTail();
  }
}
```

Example

```java
public class PettingTest
{
  public static void main(String[] args)
  {
    PetOwner jon = new PetOwner();
    Cat garfield = new Cat();
    Dog odie= new Dog();

    jon.pet(odie);
    jon.pet(garfield);
  }
}
```

Static variables and methods

- Declaring an attribute as static (only for attributes, not local variables)

  ```
  modifier static type identifier;
  ```

  where *modifier* is `public`, `private` or `protected`

- Declaring static methods

  ```
  modifier static type method_name(type1 arg1,
                                   type2 arg2,
                               ..., typen argn)
  {
      statements;
  }
  ```

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
  }
}
```

Static variables (contd.)

- 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
  }
}
```

Static variables (contd.)

```java
public class Dog
{
  public static int counter = 0;

  public Dog()
  {
    counter++;
  }
}
```

Static variables (contd.)

```
public class PettingTest
{
  public static void main(String[] args)
  {
    Dog d1, d2, d3;
    d1 = new Dog();
    d2 = new Dog();
    d3 = new Dog();
    System.out.println(d1.counter);
    System.out.println(d2.counter);
    System.out.println(d3.counter);
  }
}
```

Static methods

- Normal (non-static) methods represent the behaviour of objects
- Static methods are not associated with objects
- Static methods are only "services" provided by a class
- For example:
  - Math.sqrt
  - Math.pow
  - ...etc

Calling normal methods

- When calling a non-static method, the syntax is

  ```
  objectreference.method_name(arg1,arg2,...,argn)
  ```

  where variable has a reference to an object (e.g. `objectreference = new MyClass();`)

  For example:

  ```
  String title = new String("Lock, Stock");
  int size = title.length();
  char initial = title.charAt(0);
  ```

McGill

Calling static methods

- When calling a static method, the syntax is

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

  Forexample:

```
double power = Math.pow(2.0, 3);
```

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)

Example (contd.)

```
public class B
{
    public static void main(String[] args)
    {
        A.q();              // Prints Good bye
        A x = new A();      // Creates an A object
        x.p();              // Prints Hello
        A.p();              // Compile-time Error
        x.q();              // Prints Good bye
    }
}
```

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
  }
}
```

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(this.n); //OK
  }
  static void q()
  {
    System.out.println(this.n); //WRONG
  }
}
```

Static methods access

- A static method can be called from a non-static context, but...

- A non-static method cannot be called from a static context, because in order to call a non-static method, you need to provide a reference to an object.

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 static methods from non-static methods

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

Accessing static methods from non-static methods

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

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.

Accessing non-static methods from static methods

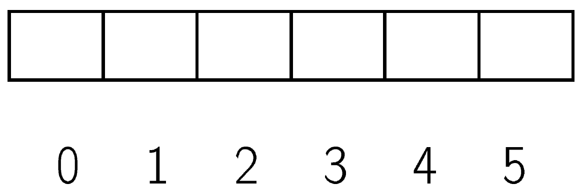
```
public class A
{
    void p()
    {
        System.out.println(‘‘Hello’’);
    }
    static void q()
    {
        System.out.println(‘‘Good bye’’);
        this.p();
    }
}
```

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.

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:

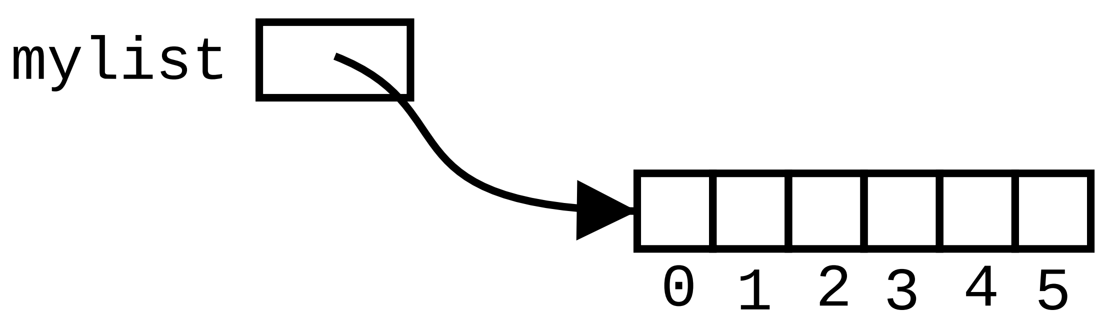
```
Mouse[] mouse_list;
```

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.

Example

```
double[] table;
table = new double[5];
table[0] = 3.141;
table[1] = 1.618;
table[2] = table[0] + table[1];
table[4] = table[2];
table[3] = 1;
table[0] = 1.414;
int i = 0;
while (i < table.length)
{
    System.out.println( table[i] );
    i++;
}
```

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] = 100.0;
    index++;
  }
}
```

Example 1

0	1	2	3	4
0.0	0.0	0.0	0.0	0.0

↑

Example 1

0	1	2	3	4
100.0	0.0	0.0	0.0	0.0
	↑			

Example 1

0	1	2	3	4
100.0	100.0	0.0	0.0	0.0
		↑		

Example 1

0	1	2	3	4
100.0	100.0	100.0	0.0	0.0
			↑	

Example 1

0	1	2	3	4
100.0	100.0	100.0	100.0	0.0
				↑

Example 1

0	1	2	3	4
100.0	100.0	100.0	100.0	100.0

↑

Example 2

- Filling an array

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

Example 1

0	1	2	3	4
0.0	1.0	4.0	9.0	16.0

$\uparrow$

Example 3

- Finding the minimum number in an array

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

Example 1

0	1	2	3	4
7.0	2.0	4.0	1.0	6.0

↑

minimum | 7.0 |

Example 1

0	1	2	3	4
7.0	2.0	4.0	1.0	6.0
	↑			

minimum | 2.0 |

Example 1

0	1	2	3	4
7.0	2.0	4.0	1.0	6.0
		↑		

minimum | 2.0 |

Example 1

0	1	2	3	4
7.0	2.0	4.0	1.0	6.0
			↑	

minimum 1.0

Example 1

0	1	2	3	4
7.0	2.0	4.0	1.0	6.0
				↑

minimum 1.0

Example 1

0	1	2	3	4
7.0	2.0	4.0	1.0	6.0

↑

minimum 1.0

Example 4

- 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