Objects as first class values

```
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);
  }
}
```

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

```
class MoviePair
{
  Movie m1, m2;

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

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

  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();
  }
}
```

Objects as first class values

- 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

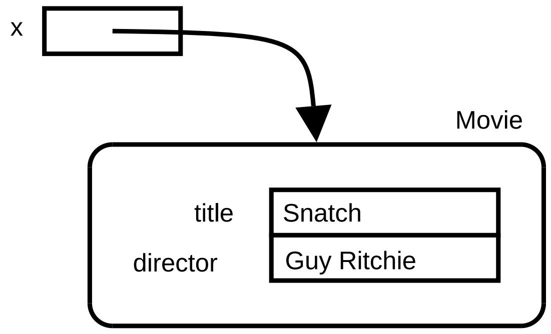
```
public class MovieApplication
{
  public static void main(String[] args)
  {
    Movie a, b;
    MoviePair pair;
    a = new Movie(“Lawrence of Arabia”, “David Lea
    b = new Movie(“Snatch”, “Guy Ritchie”);
    pair = new MoviePair();
    pair.set_first(a);
    pair.set_second(b);
    pair.play_both();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

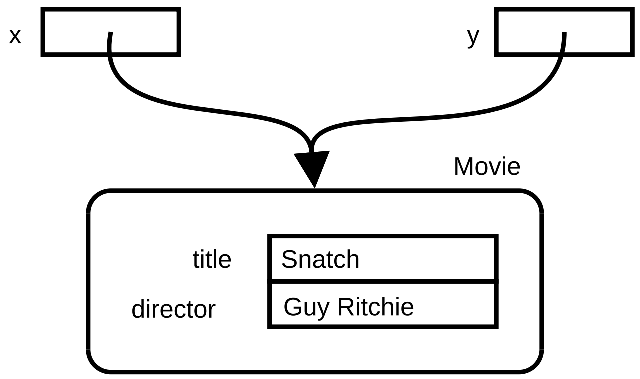


References and Aliases

- 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

```
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;
  }
  double getAccount() { return account; }
}
```

References and Aliases

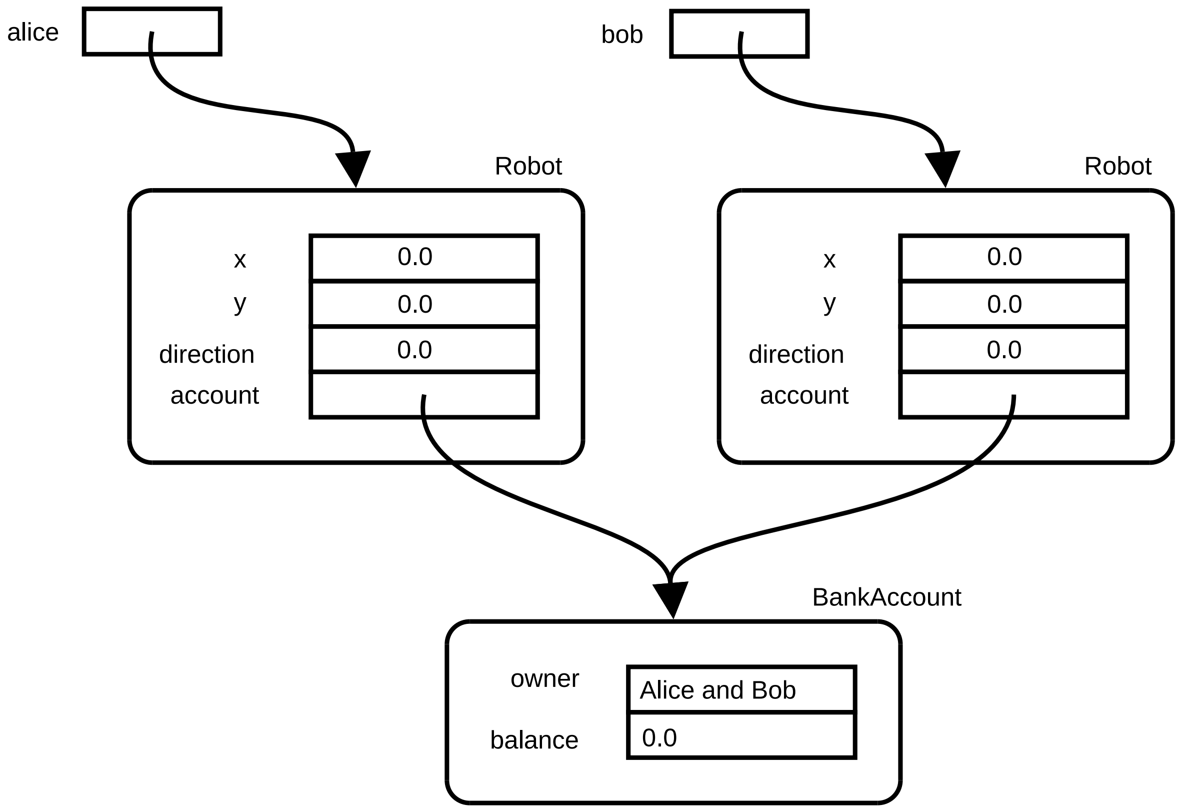
- Aliases can be used to represent shared information

```
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

- Aliases can be used to represent shared information



References and Aliases

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

```
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);
    account.deposit(200.0);
  }
}
```

References and Aliases

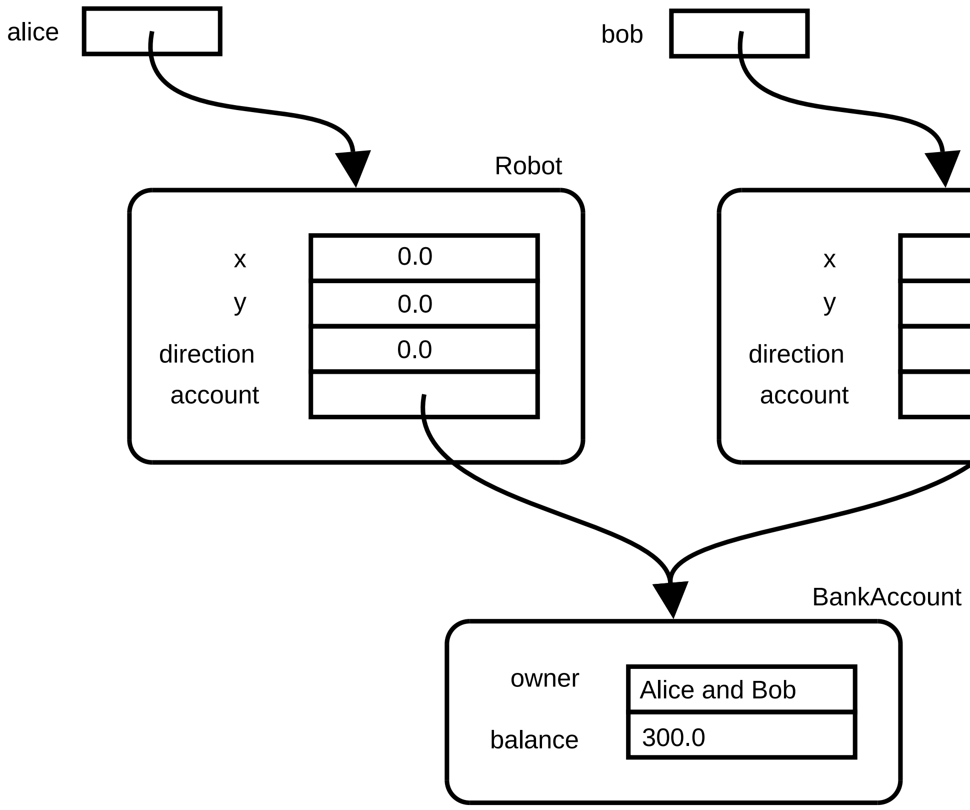
- When information is shared, changes to it affect all those who share it:

```
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



References and Aliases

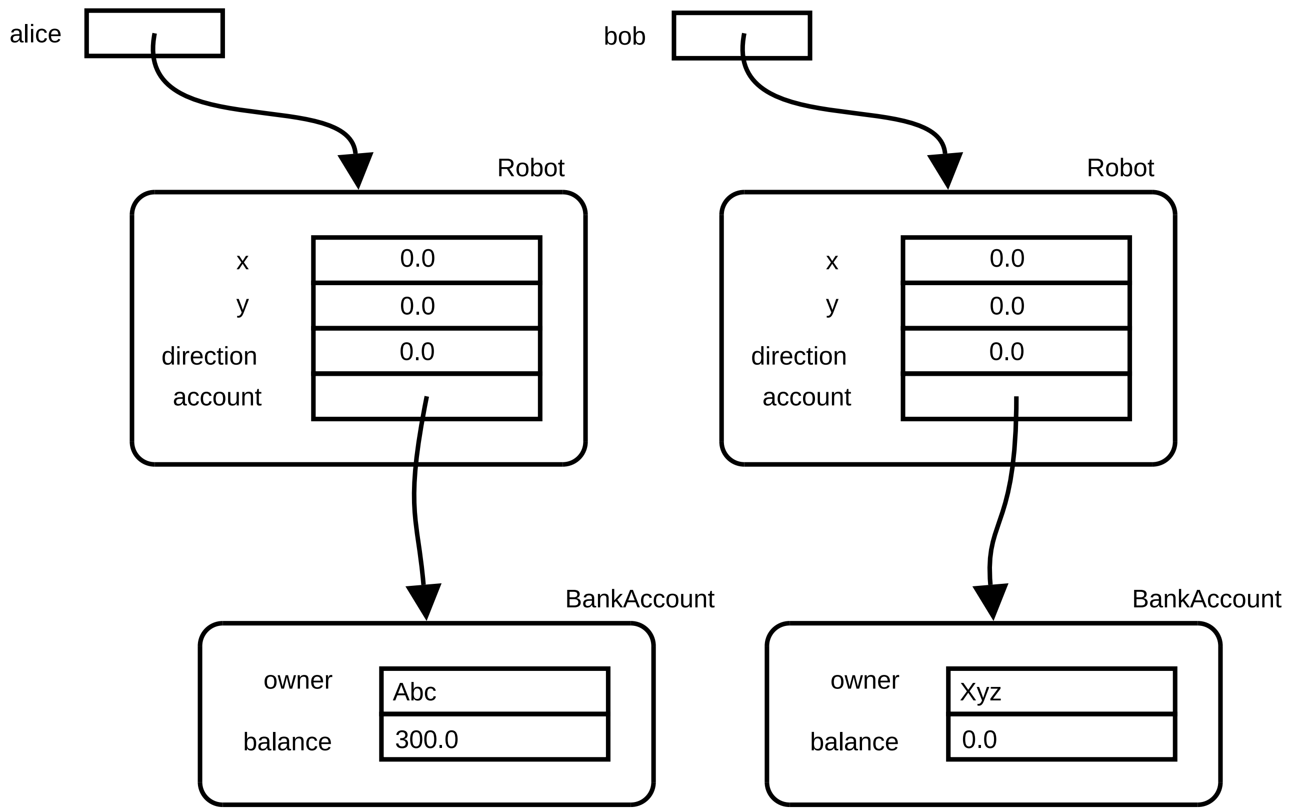
- Contrast with:

```
Robot alice, bob;
BankAccount account;

alice = new Robot(0.0);
bob = new Robot(0.0);
account1 = new BankAccount(“Abc”);
account2 = new BankAccount(“Xyz”);
alice.setAccount(account1);
bob.setAccount(account2);

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

References and Aliases



Encapsulation and visibility

- Abstraction and visibility
- Purpose of encapsulation:
  - Hiding the state of an object, (part of) the structure of an object (attributes and/or methods,) or the internal representation of data, so that a client doesn't have to know about the internals of an object (abstraction.)
  - Security: maintaining the integrity of data. Enforcing limited visibility so that clients cannot "corrupt" the state of an object, so that only the class of the object can change the object's state.
- Visibility modifiers (for attributes and methods): `public`, `private` and `protected`.
- Visibility modifiers are orthogonal (independent) of whether the attribute or method is static or not. So they can be combined in any way.

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 ar
  {
    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 CanadianDollars
{
  double amount, rate;

  CanadianDollars(double a)
  {
    amount = a;
    rate = 0.75;
  }

  double USvalue()
  {
    return amount * rate;
  }
}
```

Example

```
public class Euros
{
  double amount, rate;

  Euros(double a)
  {
    amount = a;
    rate = 1.24;
  }

  double USvalue()
  {
    return amount * rate;
  }
}
```

Example

```
public class MagicAccount
{
  double balance;

  MagicAccount() { balance = 0.0; }

  void deposit(CanadianDollars amount)
  {
    balance = balance
            + amount.USvalue() + 200.0;
  }
  void deposit(Euros amount)
  {
    balance = balance
            + amount.USvalue() + 100.0;
  }
}
```

Example

```
public class BankingApplication
{
  public static void main(String[] args)
  {
    MagicAccount b = new MagicAccount();
    CanadianDollars dollars;
    Euros eus;

    dollars = new CanadianDollars(300.0);
    eus = new Euros(100.0);
    b.deposit(dollars);
    b.deposit(eus);
  }
}
```

Static variables and methods

- Declaring an instance variable

  `type identifier`;

- Declaring visibility of the attribute

  `modifier type identifier`;

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

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

  `modifier` `static` *`type identifier`*;

Declaring methods

- Declaring normal methods

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

- Declaring static methods

```
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 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
  }
}
```

McGill

Static variables (contd.)

```
public class BankAccount
{
  private static 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;
  }
}
```

McGill

Static variables (contd.)

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

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);
  ```

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)

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

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.

The end