object
oriented
programming
#45923991

## Object Oriented MatLab

- In 2008 MatLab (R2008a) introduced an entirely new Object Oriented (OO) framework and syntax, bringing it more in line with Java, Python and C++. [We will ignore the earlier version of OO in Matlab.]

- Our description of OO is based on Java. MatLab closely follows Java in capabilities but there are lots of differences. Following our OO description, we will show an example Java program and its re-implementation in MatLab as an example. Object Oriented languages have **classes** which encapsulate data and the methods (functions) that work on them. Instances of each class with specific data values are called **objects**. For example, **BankAccount** might be a class defined to describe a bank account (with variables for account number and balance, etc) and methods defining operations on this data. Instances of BankAccount are called BankAccount **objects** and give values for individual bank accounts. For example, an instance of **BankAccount** can have account number 654321 and balance 100 and methods **deposit()** and **withdraw()** to operate on this data.

- Object Oriented programming enforces 3 behaviours:

  1. **Encapsulation:** the use of self-contained **methods** (functions) that keep the processing functions and the data they manipulate together. The data for object 654321/100 and the methods operating on this data are maintained together.

2. **Inheritance:** passes data down a hierarchy of classes. This minimizes programming efforts when adding functions to complex systems. Thus a class **ChequingAccount** could be isa (extends) **BankAccount** and thus inherits **BankAccount**'s variables and methods, while adding new variables, say, **transactionCount**, and new or modified methods that operate on this new data, for our example, refined versions of **deposit()** and **withdraw()** are used. Only the methods and data associated with **ChequingAccount** need be added, all the methods and data for **BankAccount** already exist and can be used in **ChequingAccount**.

   – An instance of **ChequingAccount** might have account number 123456, balance 1000 and transaction count 5. Inheritance's ability to re-use existing classes is a major advantage of object oriented programming. Inheritance allows us to follow the **Abstraction** paradigm: writing code in the most generic class possible and specializing as needed.

3. **Polymorphism:** a programmer can create methods for objects, whose exact class is not known until runtime. (It appears Polymorphism is not directly support by MatLab OO but there are known workarounds.)

- Is class A isa class B, then B is the parent (super) class of A and A is the child (sub) class of B. Subclasses inherit from superclasses but not the other way around.

- Classes have **constructor** methods (functions) that are executed when an instance of a class, i.e. object, is created. This method might also do some initial assignments to that object's variables.

- Traditional, procedural/functional programs are often "long" list of commands, perhaps **decomposed** into smaller reusable groups of these statements (called functions or subroutines) that can be executed multiple times. Each function might perform a particular task. This type of programming allows a program's data to be accessed from any other part of the program. When these programs grow in size, allowing any function to modify any piece of data means that bugs can have wide-reaching effects.

- On the other hand, object oriented programming encourages the programmer to place data where it can not be directly accessible by the rest of the program You must use an object's **getter** and **setter** methods to retrieve or modify this data in Java, the internal variable names and structures are hidden and unknown.

- For example, we can have getter methods **getAccountNumber()** and **getBalance()** for **BankAccount** and an additional getter method **getTransactionCount()** in **ChequingAccount** and setter methods like **setAccountNumber()** and **setBalance()** for **BankAccount** and an additional getter method **setTransactionCount()** in **ChequingAccount**.

- These methods allow you to obtain the values of **private/protected** variables of objects without knowing their names. These methods act as the intermediaries for retrieving or modifying the data they control. This prevents bugs from having far reaching effects.

- You can also use getter and setter methods in MatLab but one does not necessarily have to use **getter** and **setter** methods like in Java. You can can use object.method() syntax directly, for example, BA.balance gives the balance for BankAccount object BA.

- Objects can be thought of as **encapsulating** their data with set of methods designed to ensure that the data are used appropriately and to assist in that use. The object's methods typically include checks and safeguards specific to the data types the object contains.

- An object can also offer simple-to-use, standardized methods for performing particular operations on its data, while concealing the specifics of how those tasks are accomplished. In this way, alterations can be made to the internal structure or methods of an object without requiring that the rest of the program be modified. This approach can also be used to offer standardized methods across different types of objects.

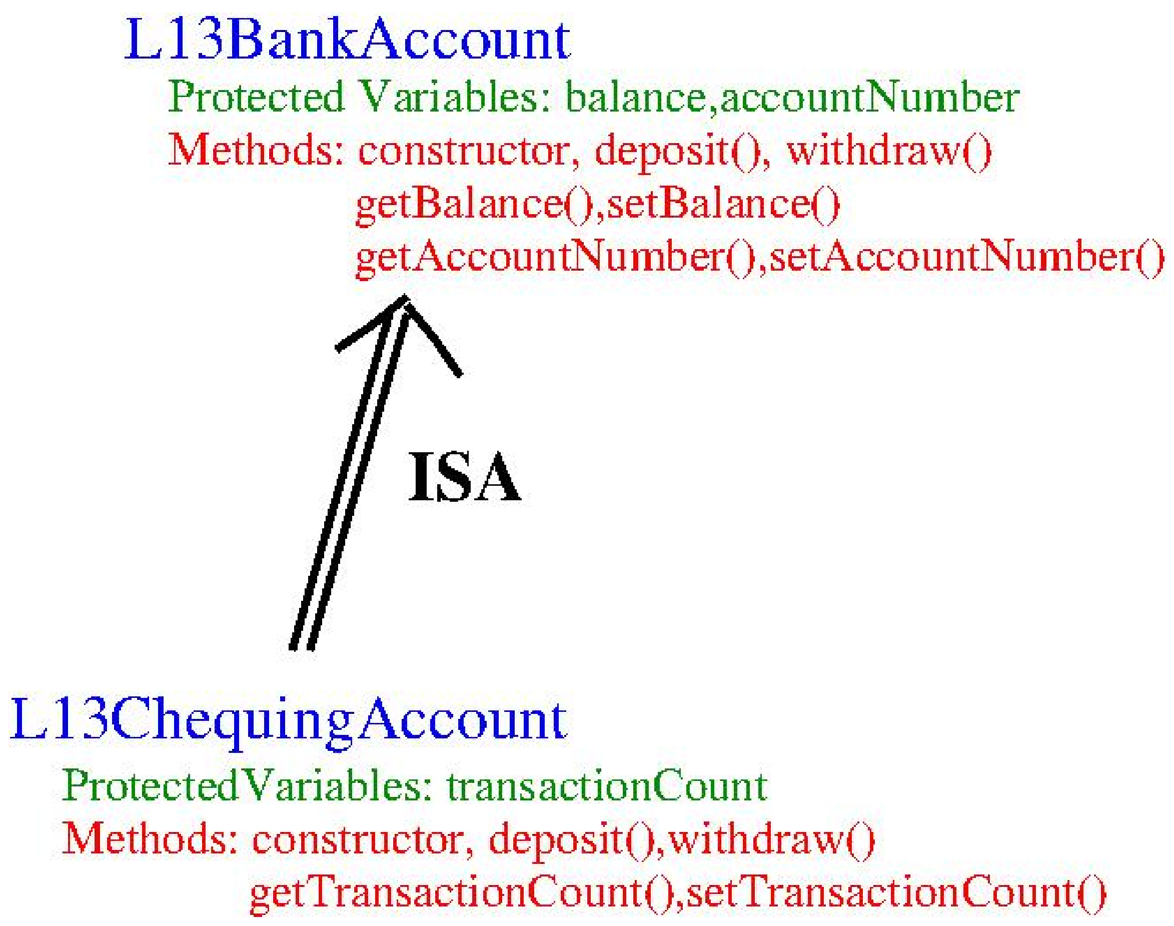
- As an example, several different types of objects might offer print methods (**toString** in Java actually return a character string for printing purposes). Each type of object might implement that print method in a different way, reflecting the different kinds of data each contains, but all the different print methods might be called in the same standardized manner from elsewhere in the program.

- This type of **abstraction** become especially useful when more than one programmer is contributing code to a project or when the goal is to reuse code between projects. For example, in CS1027, student learn how to write classes for stacks and queues that are abstract data types (ADTs) and can be used for objects of any type.

- Operations on objects are defined by methods that **encapsulate** data and new or existing methods. One can refine existing methods by **overloading** or **overriding** them. Overloaded methods have the same name as one or more existing methods but the number and corresponding types of its parameters distinguish it from the other methods. If a method with an existing name has the same number and corresponding type of parameters as an existing method, it overrides the existing method. In Java, such methods must be in a isa class hierarchy.

## Java/MatLab BankAccount Example

- The figure below shows in more detail the **BankAccount** and its subclass **ChequingAccount** classes outlining their data and methods. [We prefix the names of these classes with **L13** to indicate that they belong to this lecture. All L13*m or L13*java files are MatLab or Java code for this lecture.]


L13BankAccount
Protected Variables: balance,accountNumber
Methods: constructor, deposit(), withdraw()
getBalance(),setBalance()
getAccountNumber(),setAccountNumber()
ISA
L13ChequingAccount
ProtectedVariables: transactionCount
Methods: constructor, deposit(),withdraw()
getTransactionCount(),setTransactionCount()

The **L13BankAccount** and **L13ChequingAccount** classes. **L13ChequingAccount** isa (extends) the **L13BankAccount**, **L13ChequingAccount** is **L13BankAccount**'s subclass and inherits its variables and methods, perhaps refining the methods by overloading or overriding. **L13BankAccount** is **L13ChequingAccount**'s **super** class (variables and methods in **L13ChequingAccount** are not available to **L13BankAccount**.

## The Java Code

- We present a Java program below that does simple manipulation of a **L13BankAccount** class and its child class **L13ChequingAccount** (realized as 3 Java files, **L13BankAccount.java**, **L13ChequingAccount.java** and **TestAccount.java**).
- The **L13BankAccount.java** file:

```
public class L13BankAccount {

// protected allows variables in subclass in an ISA
// hierarchy to access variables in their super classes
protected int accountNumber;
protected int balance;

public L13BankAccount(int accountNumber,int balance) {
this.accountNumber=accountNumber;
this.balance=balance;
System.out.format("The constructor in class L13BankAccount executed\n");
System.out.format("L13BankAccount object with account number=%d\n",
                   this.accountNumber);
System.out.format("and balance=%d created\n",
                   this.balance);
}

public void deposit(int amount) {
this.balance+=amount;
System.out.format("A deposit made using method deposit()\n");
System.out.format("in class L13BankAccount\n") ;
System.out.format("amount=%d balance=%d\n",amount,this.balance);
}

public void withdraw(int amount) {
this.balance-=amount;
System.out.format("A withdrawl made using method withdraw()\n");
System.out.format("in class L13BankAccount\n") ;
System.out.format("amount=%d balance=%d\n",amount,this.balance);
}

// getters and setters
public int getBalance() {
return this.balance;
}

public void setBalance(int balance) {
this.balance=balance;
System.out.format("balance set to %d in setter ",this.balance);
System.out.format("setBalance() in class L13BankAccount\n");
}
```

```
42
43 public int getAccountNumber() {
44 return this.accountNumber;
45 }
46
47 public void setAccountNumber(int accountNumber) {
48 this.accountNumber=accountNumber;
49 System.out.format("account number set to %d in setter ",
                     this.accountNumber);
50 System.out.format("setAccount() in class L13BankAccount");
51 }
52
53 } // class L13BankAccount
```

- Some discussion on **L13BankAccount.java**:

  1. The keyword **protected** in lines 5 and 6 mean the variables (**accountNumber** and **balance**) can be seems in any of the subclasses of **L13BankAccount.java**.

  2. The constructor method **L13BankAccount** (lines 8 to 16) had the

same name as the class and sets the value of protected variables **accountNumber** and **balance**.

3. The keyword **this** in lines 9 and 10 mean the “current” object. This means **this.balance=balance** causes the value of parameter **balance** to be assigned to protected variable **balance**.
4. Methods **deposit()** and **withdraw** increment or decrement the protected variable **balance**.
5. The getter and setter methods do the expected thing for protected variables **accountNumber** and **balance**.

- The **L13ChequingAccount.java** file:

```
public class L13ChequingAccount extends L13BankAccount  {

private int transactionCount;

// constructor method for L13ChequingAccount class with inheritance
// from L13BankAccount class
public L13ChequingAccount(int accountNumber,int balance) {
// call constructor of super class, L13BankAccount
super(accountNumber,balance);
System.out.format("Super L13BankAccount constructor method ");
System.out.format("finished execution\n");
this.transactionCount=0;
System.out.format("The constructor method in class ");
System.out.format("L13ChequingAccount executed\n");
System.out.format("L13ChequingAccount object created ");
System.out.format("with account number=%d, balance=%d\n",
         this.accountNumber,this.balance);
System.out.format("and transaction count=%d created\n\n",
         this.transactionCount);
}

public void deposit(int amount) {
// use the deposit method in the super class, L13BankAccount
super.deposit(amount);
System.out.format("Super L13BankAccount method deposit() ");
System.out.format("finished execution\n");
```

```
this.transactionCount+=1;
System.out.format("A deposit made using method deposit() ");
System.out.format("in class L13ChequingAccount\n") ;
// Use getters
System.out.format("amount=%d balance=%d transaction count=%d\n",
                   amount,this.getBalance(),this.getTransactionCount());
}

public void withdraw(int amount) {
// use the withdraw method in the super class, L13BankAccount
super.withdraw(amount);
System.out.format("Super L13BankAccount method withdraw() ");
System.out.format("finished execution\n");
this.transactionCount+=1;
System.out.format("A withdrawl made using method withdraw() ");
System.out.format("in class L13ChequingAccount\n") ;
// Use private variables
System.out.format("amount=%d balance=%d transaction count=%d\n",
                   amount,this.balance,this.transactionCount);
}

// getters and setters
public int getTransactionCount() {
return this.transactionCount;
}

public void setTransactionCount(int transactionCount) {
this.transactionCount=transactionCount;
```

```
55 System.out.format("transactionCount set to %d in setter\n",
56                    this.transactionCount);
57 System.out.format("setTransactionCount() in class L13ChequingAccount\n");
58 }
59
60 } // class L13ChequingAccount
```

- Some discussion on **L13ChequingAccount.java**:

  1. The keyword **super** on line 9 means call the constructor class of the superclass of **L13ChequingAccount**. Thus the constructor class in **L13BankAccount** is executed. In addition to doing this, the constructor method **L13ChequingAccount** sets the protected variable **transactionCount**.

  2. Similarly, in the overridden methods **deposit()** and **withdraw()** the **deposit()** and **withdraw()** methods in the superclass **L13BankAccount**

are executed and then the protect variable **transactionCount** is incremented by 1.

3. There are new getter and setter methods for protected variable **transactionCount**.

- The **L13TestAccount.java** file:

```
public class L13TestAccount {

// static main method is initially executed
public static void main(String[] args) {
System.out.format("\nJAVA\n");

System.out.format("\nCreate L13BankAccount object\n");
System.out.format("============================\n");
L13BankAccount yourAccount = new L13BankAccount(654321,100);

System.out.format("\nCreate L13ChequingAccount object\n");
System.out.format("================================\n");
L13ChequingAccount myAccount = new L13ChequingAccount(123456,70);

System.out.format("\nTest L13BankAccount object\n");
System.out.format("==========================\n");
int acct=yourAccount.getAccountNumber();
yourAccount.setBalance(500);
System.out.format("\nTransactions for account: %d\n",acct);
System.out.format("--------------------------------\n");
yourAccount.deposit(20);
System.out.format("\n");
yourAccount.deposit(80);
System.out.format("\n");
yourAccount.withdraw(50);

System.out.format("\nTest L13ChequingAccount object\n");
System.out.format("===============================\n");
acct=myAccount.getAccountNumber();
myAccount.setBalance(1000);
myAccount.setTransactionCount(5);
System.out.format("\nTransactions for account: %d\n",acct);
System.out.format("--------------------------------\n");
myAccount.deposit(20);
System.out.format("\n");
myAccount.withdraw(50);
}

} // L13TestAccount
```

- Some discussion on **L13TestAccount.java**:

  1. The **main** method is declared to be public and static (so it can exist independently of an object). We compile all 3 java files in command line mode by using **javac L13TestAccount.java**. Java will detect that **L13BankAccount** and **L13ChequingAccount** are used

by **L13TestAccount** and compile them for you. Then we run the code using **java L13TestAccount**. This causes the **main** method in **L13TestAccount** to be executed.

2. Line 9 shows the instantiation of an object **yourAccount** using the keyword **new**. The constructor for class **L13BankAccount** is executed with parameters 654321 and 100.
3. Similarly, a new object **myAccount** is instantiated on line 13 with parameters 123456 and 70. What happened to **transactionCount**? It is initialized to 0.
4. Lines 15 to 25 and lines 27 to 36 then use the various methods in the 2 classes for objects **yourAccount** and **myAccount**.

## The MatLab Code

- We present an Object Oriented MatLab below that does exactly the same thing as the above Java program (realized as 3 MatLab files, **L13BankAccount.m**, **L13ChequingAccount.m** and **L13TestAccount.m**).
- The program also has classes **L13BankAccount** and **L13ChequingAccount** but while **L13TestAccount.java** is another Java class (with a **main()** method) **L13TestAccount.m** is a simple MatLab script file. Object Oriented MatLab is simpler than Java or C++ but is missing some of Java's and C++'s features. Why program in Object Oriented MatLab? Because such a progran has access to up to 80+ MatLab toolboxes that other Object Oriented languages do not.

- The **L13BankAccount.m** file:

```
classdef L13BankAccount

% protected allows variables in subclass in an ISA
% hierarchy to access variables in their super classes
properties(SetAccess=protected)
accountNumber;
balance;
end

methods

% BA is L13BankAccount object
function BA=L13BankAccount(accountNumber,balance)
BA.accountNumber=accountNumber;
BA.balance=balance;
fprintf('The constructor in class L13BankAccount executed\n') ;
fprintf('L13BankAccount object with account number=%d\n',...
         BA.accountNumber);
fprintf('and balance=%d created\n',...
         BA.balance);
end

function BA=deposit(BA,amount)
BA.balance=BA.balance+amount;
fprintf('A deposit made using method deposit()\n');
fprintf('in class L13BankAccount\n') ;
fprintf('amount=%d balance=%d\n',amount,BA.balance);
end

function BA=withdraw(BA,amount)
BA.balance=BA.balance-amount;
fprintf('A withdrawl made using method withdraw()\n');
fprintf('in class L13BankAccount\n') ;
fprintf('amount=%d balance=%d\n',amount,BA.balance);
end

% getters and setters
function [balance]=getBalance(BA)
balance=BA.balance;
end

function BA=setBalance(BA,balance)
BA.balance=balance;
fprintf('balance set to %d in setter ',BA.balance);
fprintf('setBalance() in class L13BankAccount\n');
end

function [accountNumber]=getAccountNumber(BA)
accountNumber=BA.accountNumber;
end

function BA=setAccountNumber(BA,accountNumber)
BA.accountNumber=accountNumber;
fprintf('account number set to %d in setter ',BA.accountNumber);
```

```
55 fprintf('setAccount() in class L13BankAccount');
56 end
57
58 end % methods
59
60 end % classdef L13BankAcccount
```

- Some discussion on **L13BankAccount.m**:

  1. The constructor **L13BankAccount** on line 13 has the same name as the class. Because MatLab is a pass-by-value language, the constructor must explicitly return the object instance, in this case **BA**. Essentially, an object is a **struct** variable with associated methods.
  2. The **deposit()** and **withdraw()** methods take the object as their first parameter and return the modified object as their values. To access the protected variables, we use normal MatLab syntax like

**BA.balance=balance**. We can compare that to the Java syntax we used **this.balance=balance**.

3. The **deposit()** and **withdraw()** methods and the getter and setter methods do the same thing as the corresponding Java method does.

- The **L13ChequingAccount.m** file:

```
% The L13ChequingAccount class ISA (extends) the L13BankAccount class
classdef L13ChequingAccount < L13BankAccount

properties(Access=private)
transactionCount;
end

methods

% constructor method for L13ChequingAccount class with inheritance
% from L13BankAccount class
function CA=L13ChequingAccount(accountNumber,balance)
% call constructor of super class, L13BankAccount
CA=CA@L13BankAccount(accountNumber,balance);
fprintf('Super L13BankAccount constructor method ');
fprintf('finished execution\n');
CA.transactionCount=0;
fprintf('The constructor method in class ');
fprintf('L13ChequingAccount executed\n');
fprintf('L13ChequingAccount object created with account ');
fprintf('number=%d, balance=%d\n',CA.accountNumber,CA.balance);
fprintf('and transaction count=%d created\n\n',...
         CA.transactionCount);
end

function CA=deposit(CA,amount)
```

```
% use the deposit method in the super class, L13BankAccount
CA=CA.deposit@L13BankAccount(amount);
fprintf('Super L13BankAccount method deposit() ')
fprintf('finished execution\n');
CA.transactionCount=CA.transactionCount+1;
fprintf('A deposit made using method deposit() ');
fprintf('in class L13ChequingAccount\n') ;
% Use getters
fprintf('amount=%d balance=%d transaction count=%d\n',...
         amount,CA.getBalance(),CA.getTransactionCount);
end

function CA=withdraw(CA,amount)
% use the withdraw method in the super class, L13BankAccount
CA=CA.withdraw@L13BankAccount(amount);
fprintf('Super L13BankAccount method withdraw() ');
fprintf('finished execution\n');
CA.transactionCount=CA.transactionCount+1;
fprintf('A withdrawl made using method withdraw() ');
fprintf('in class L13ChequingAccount\n') ;
% Use private variables
fprintf('amount=%d balance=%d transaction count=%d\n',...
         amount,CA.balance,CA.transactionCount);
end

% getters and setters
function [transCt]=getTransactionCount(CA)
transCt=CA.transactionCount;
```

```
end

function CA=setTransactionCount(CA,transCt)
CA.transactionCount=transCt;
fprintf('transactionCount set to %d in setter\n',...
         CA.transactionCount);
fprintf('setTransactionCount() in class L13ChequingAccount\n');
end

end % methods
end % classdef L13ChequingAccount
```

- Some discussion on **L13ChequingAccount.m**:

  1. On line 2 the symbol < is used to indicate that class **L13ChequingAccount)** isa (or extends) class **L13BankAccount**.
  2. Line 12: the object returned by constructor method **L13ChequingAccount** is **CA**.
  3. Line 14: the constructor method for the superclass of **L13ChequingAccount** in **L13BankAccount** is executed. Note the syntax **CA@L13BankAccount** is correct as **L13BankAccount** is a method for the **L13ChequingAccount** class.
  4. The **deposit()** and **withdraw()** methods on lines 26 and 39 respectively take **CA** as the first parameter and return the modified **CA** as the value.

5. Line 28 **CA=CA.deposit@L13BankAccount()** and line 41 **CA=CA.withdraw@BankAccount** are equivalent to the super statements in the **deposit()** and **withdraw()** methods in **L13ChequingAccount.java**.
6. There is a getter and a setter for **transactionCount**.

- The **L13TestAccount.m** file:

```
% MatLab script file L13TestAccount.m
fprintf('\nMATLAB\n');

fprintf('\nCreate L13BankAccount object\n');
fprintf('============================\n');
yourAccount=L13BankAccount(654321,100);

fprintf('\nCreate L13ChequingAccount object\n');
fprintf('================================\n');
myAccount=L13ChequingAccount(123456,70);

fprintf('\nTest L13BankAccount object\n');
fprintf('==========================\n');
acct=yourAccount.getAccountNumber();
yourAccount=yourAccount.setBalance(500);
fprintf('\nTransactions for account: %d\n',acct);
fprintf('--------------------------------\n');
yourAccount=yourAccount.deposit(20);
fprintf('\n');
yourAccount=yourAccount.deposit(80);
fprintf('\n');
yourAccount=yourAccount.withdraw(50);

fprintf('\nTest L13ChequingAccount object\n');
fprintf('==============================\n');
acct=myAccount.getAccountNumber();
```

```
myAccount=myAccount.setBalance(1000);
myAccount=myAccount.setTransactionCount(5);
fprintf('\nTransactions for account: %d\n',acct);
fprintf('--------------------------------\n');
myAccount=myAccount.deposit(20);
fprintf('\n');
myAccount=myAccount.withdraw(50);

% L13TestAccount.m
```

- Some discussion on **L13TestAccount.m**:

  1. This code s a simple MatLab script.
  2. Variables **yourAccount** and **myAccount** are the objects generated by the 2 class constructor methods.
  3. We used the getter and setters for the 2 classes as in Java. But instead of **yourAccount.setBalance(500** we could have used **yourAccount.balance=500**.

- Both programs produce identical output (see **L13java_output.java** and **L13matlab_output.txt**), which we show below.

```
JAVA or MATLAB

Create L13BankAccount object
============================
The constructor in class L13BankAccount executed
L13BankAccount object with account number=654321
and balance=100 created

Create L13ChequingAccount object
================================
The constructor in class L13BankAccount executed
L13BankAccount object with account number=123456
and balance=70 created
Super L13BankAccount constructor method finished execution
The constructor method in class L13ChequingAccount executed
L13ChequingAccount object created with account number=123456, balance=70
and transaction count=0 created

Test L13BankAccount object
==========================
balance set to 500 in setter setBalance() in class L13BankAccount

Transactions for account: 654321
```

```
--------------------------------
A deposit made using method deposit()
in class L13BankAccount
amount=20 balance=520

A deposit made using method deposit()
in class L13BankAccount
amount=80 balance=600

A withdrawl made using method withdraw()
in class L13BankAccount
amount=50 balance=550

Test L13ChequingAccount object
==============================
balance set to 1000 in setter setBalance() in class L13BankAccount
transactionCount set to 5 in setter
setTransactionCount() in class L13ChequingAccount

Transactions for account: 123456
--------------------------------
A deposit made using method deposit()
in class L13BankAccount
amount=20 balance=1020
Super L13BankAccount method deposit() finished execution
A deposit made using method deposit() in class L13ChequingAccount
amount=20 balance=1020 transaction count=6
```

```
A withdrawl made using method withdraw()
in class L13BankAccount
amount=50 balance=970
Super L13BankAccount method withdraw() finished execution
A withdrawl made using method withdraw() in class L13ChequingAccount
amount=50 balance=970 transaction count=7
```

## Using Predefined Object Oriented MatLab Classes

- Most MatLab users will not be writing object oriented code but rather using MatLab supplied object oriented classes.
- We demonstrate MatLab supplied Object Oriented classes using the VideoWriter class. The example involves the generation of a video of a fluoroscopic images of a bending knee

## Bending Knee Video

- Consider the following MatLab code in **L13_make_bending_knee_video_with_OO.m** which generates an avi video from fluoroscopic jpeg images of a bending knee:

```
% Create a video object
outputVideo = VideoWriter('L13_bending_knee_video.avi');
% set that object's FrameRate field to 5
outputVideo.FrameRate = 5;
open(outputVideo);

pathname='.';
% read 21 jpg images from 70 to 90 and save
% these images to the video object
for i = 70:90
    stg=sprintf('%03d',i);
    filename=['Brox-levels-1-eta-0.95-outer-10-inner-10.'...
               stg '.jpg'];
    fprintf('Processing file: %s\n',filename);
    img = imread([pathname '/' filename]);
    writeVideo(outputVideo,img);
end
```

```
close(outputVideo);
fprintf('L13_bending_knee_video.dvi created\n');
% display the avi file
implay('L13_bending_knee_video.avi');
```

- The statement:

  `outputVideo = VideoWriter('L13_bending_knee_video.avi')`

  creates a video object using the **constructor** `VideoWriter` method and puts the address (pointer or reference value) of the object in the variable `outputVideo`.

- The statement: `outputVideo.FrameRate = 5` sets the `FrameRate` variable of object `outputVideo` to 5.

- The statement: `open(outputVideo)` opens the object for images to be added.

- The statement: `writeVideo(outputVideo,img)` adds a jpeg image in `img` to this object.

- The statement: `close(outputVideo)` closes this object so no further images can be added.

- Finally, the statement:

  ```
  implay('L13_bending_knee_video.avi')
  ```

  plays the image sequence using the file

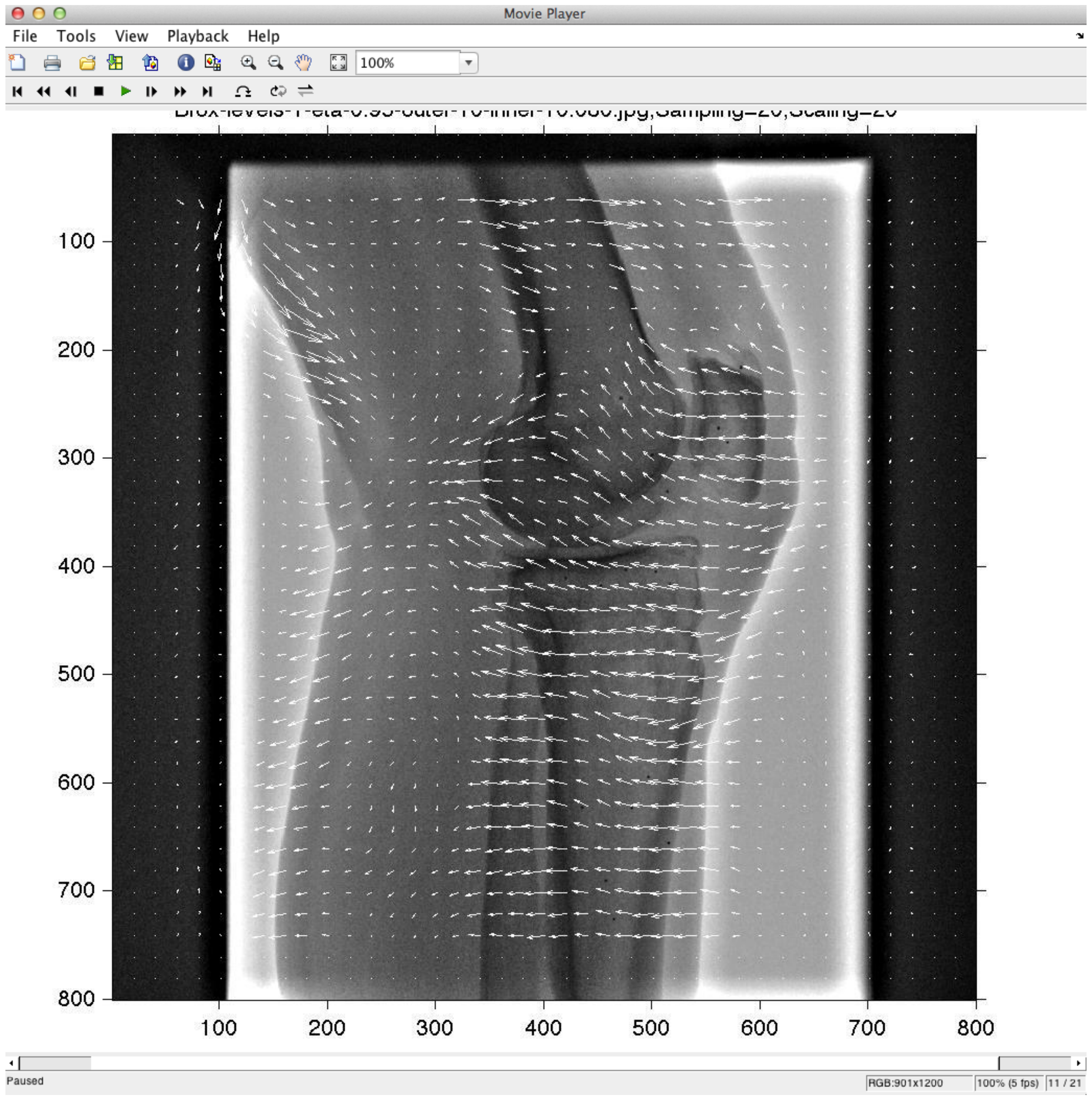
  ```
       L13_bending_knee_video.avi.
  ```

- `implay('L13_bending_knee_video.avi')` plays this 21 frame video at 5 frames per second.

- This function produces the following output:

```
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.070.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.071.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.072.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.073.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.074.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.075.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.076.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.077.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.078.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.079.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.080.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.081.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.082.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.083.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.084.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.085.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.086.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.087.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.088.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.089.jpg
Processing file: Brox-levels-1-eta-0.95-outer-10-inner-10.090.jpg
L13_video.dvi created
```

- The figure below shows the middle frame of this sequence (Brox-levels-1-eta-0.95-outer-10-inner-10.080.jpg):



The middle frame of the L13_bending_knee_video.avi video. This movie shows the optical flow at sub-sampled pixels in the images indicating the 2D motion of each pixel.