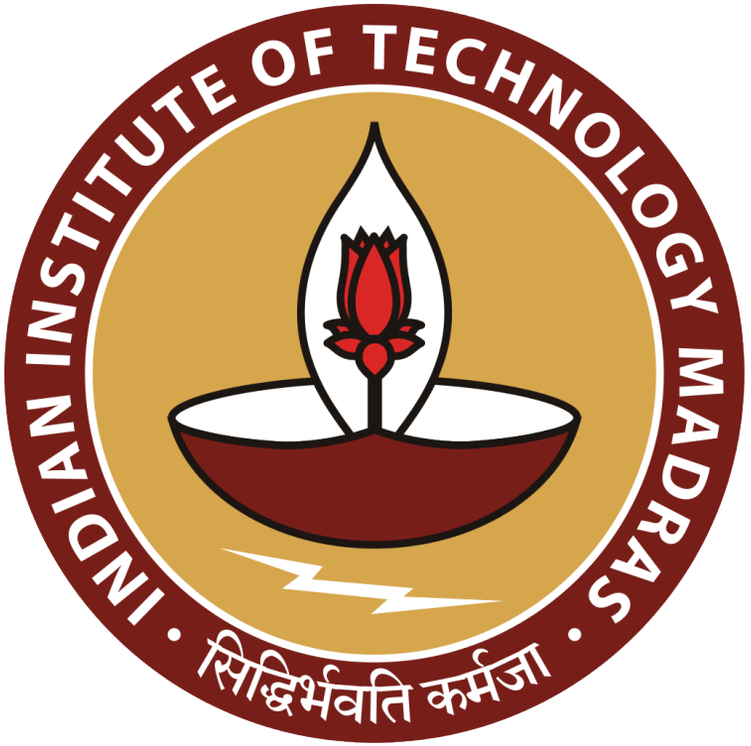

IIT Madras

ONLINE DEGREE

Encapsulation

What is encapsulation?

- Procedures that we have seen so far have been unanchored.
  - It seems more natural to attach the procedure to the data elements on which it operates

What is encapsulation?

- Procedures that we have seen so far have been unanchored.
  - It seems more natural to attach the procedure to the data elements on which it operates
- **Encapsulate** the data elements and the procedures into one package
  - which is called an **object**, hence the popular term **object oriented** computing/programming

What is encapsulation?

- Procedures that we have seen so far have been unanchored.
  - It seems more natural to attach the procedure to the data elements on which it operates
- **Encapsulate** the data elements and the procedures into one package
  - which is called an **object**, hence the popular term **object oriented** computing/programming
- The procedures encapsulated in the object act as the interfaces through which the external world can interact with the object

What is encapsulation?

- Procedures that we have seen so far have been unanchored.
  - It seems more natural to attach the procedure to the data elements on which it operates
- **Encapsulate** the data elements and the procedures into one package
  - which is called an **object**, hence the popular term **object oriented** computing/programming
- The procedures encapsulated in the object act as the interfaces through which the external world can interact with the object
- The object can hide details of the implementation from the external world
  - The changed implementation may involve additional (intermediate) data elements and additional procedures
  - Any changes made to the implementation does not impact the external world, since the procedure interface is not changed

What is encapsulation?

- Procedures that we have seen so far have been unanchored.
  - It seems more natural to attach the procedure to the data elements on which it operates
- **Encapsulate** the data elements and the procedures into one package
  - which is called an **object**, hence the popular term **object oriented** computing/programming
- The procedures encapsulated in the object act as the interfaces through which the external world can interact with the object
- The object can hide details of the implementation from the external world
  - The changed implementation may involve additional (intermediate) data elements and additional procedures
  - Any changes made to the implementation does not impact the external world, since the procedure interface is not changed
- Allows for separation of concerns - separate the "what?" from the "how?"

Do we need anything beyond procedures?

- Procedures already provide some kind of encapsulation.
  - Interface via the *parameters* and *return value*
  - Hiding of variables used within the procedure

Do we need anything beyond procedures?

- Procedures already provide some kind of encapsulation.
  - Interface via the *parameters* and *return value*
  - Hiding of variables used within the procedure
- But we could have the following issues with procedures:
  - Procedures could have side-effects - some of them are desirable (recall the delete key example?)
  - We may need to call a sequence of procedures to achieve something - we expect the object to hold the state between the procedure calls (ATM example)

Datatypes and encapsulation

- Recall datatypes?
  - Basic datatypes - integer, character, boolean
  - Subtype of a datatype to restrict the values and operations
  - Records, Lists, Strings: compound datatypes
  - Each card (table row) can be represented using a datatype
  - The collection of cards can also be represented as a datatype

Datatypes and encapsulation

- Recall datatypes?
  - Basic datatypes - integer, character, boolean
  - Subtype of a datatype to restrict the values and operations
  - Records, Lists, Strings: compound datatypes
  - Each card (table row) can be represented using a datatype
  - The collection of cards can also be represented as a datatype
- Operations on a datatype
  - Well defined operations on the basic datatypes and their subtypes
  - Some operations on the string datatype

Datatypes and encapsulation

- Recall datatypes?
  - Basic datatypes - integer, character, boolean
  - Subtype of a datatype to restrict the values and operations
  - Records, Lists, Strings: compound datatypes
  - Each card (table row) can be represented using a datatype
  - The collection of cards can also be represented as a datatype
- Operations on a datatype
  - Well defined operations on the basic datatypes and their subtypes
  - Some operations on the string datatype
- But what about operations on compound datatypes?

Datatypes and encapsulation

- Recall datatypes?
  - Basic datatypes - integer, character, boolean
  - Subtype of a datatype to restrict the values and operations
  - Records, Lists, Strings: compound datatypes
  - Each card (table row) can be represented using a datatype
  - The collection of cards can also be represented as a datatype
- Operations on a datatype
  - Well defined operations on the basic datatypes and their subtypes
  - Some operations on the string datatype
- But what about operations on compound datatypes?
- What if we are allowed to attach our own procedures to a datatype?
  - Allow a datatype to have procedure fields
  - Just as X.F represents the field F of X, we can use X.F(a,b) to represent a procedure field of X which takes parameters a and b.

Example: Classroom Scores dataset

- Questions most frequently asked of the Scores dataset
  - What is the average in a subject - say Maths?
  - What is the overall average?

Example: Classroom Scores dataset

- Questions most frequently asked of the Scores dataset
  - What is the average in a subject - say Maths?
  - What is the overall average?
- Suppose that we have to ask these questions many times
  - It is wasteful to carry out the same computation again and again
  - Can we not store the answer of the question, and just return the saved answer when the question is asked again ?

Example: Classroom Scores dataset

- Questions most frequently asked of the Scores dataset
  - What is the average in a subject - say Maths?
  - What is the overall average?
- Suppose that we have to ask these questions many times
  - It is wasteful to carry out the same computation again and again
  - Can we not store the answer of the question, and just return the saved answer when the question is asked again ?
- This will work as long as the dataset is static.

Example: encapsulation

- We could create an object CT (for class teacher) of datatype ClassAve
  - ClassAve needs only the list of total marks of the students

Example: encapsulation

- We could create an object CT (for class teacher) of datatype ClassAve
  - ClassAve needs only the list of total marks of the students
- ClassAve can have a field marksList that holds the total marks of all the students in the classroom dataset.

Example: encapsulation

- We could create an object CT (for class teacher) of datatype ClassAve
  - ClassAve needs only the list of total marks of the students
- ClassAve can have a field marksList that holds the total marks of all the students in the classroom dataset.
- We can now add a procedure average() to ClassAve to find the average of the list

Example: encapsulation

- We could create an object CT (for class teacher) of datatype ClassAve
  - ClassAve needs only the list of total marks of the students
- ClassAve can have a field marksList that holds the total marks of all the students in the classroom dataset.
- We can now add a procedure average() to ClassAve to find the average of the list
- Since we want to store the answer after the first time, we could have another field aValue that will hold the computed value of average. Initially aValue = -1.

Example: encapsulation

- We could create an object CT (for class teacher) of datatype ClassAve
  - ClassAve needs only the list of total marks of the students
- ClassAve can have a field marksList that holds the total marks of all the students in the classroom dataset.
- We can now add a procedure average() to ClassAve to find the average of the list
- Since we want to store the answer after the first time, we could have another field aValue that will hold the computed value of average. Initially aValue = -1.
  - CT.average() first checks if CT.aValue is -1.
  - If no, it just returns CT.aValue
  - If it is -1, this means that the average has not been computed yet. So, it computes the average by summing the marks in the list and dividing the sum by the length of the list

Example: encapsulation

- What about the average values of the individual subjects?

Example: encapsulation

- What about the average values of the individual subjects?
- Just like CT, we could have objects PhT, MaT and ChT (for Physics, Maths and Chemistry Teachers) that each hold (or have access to) the entire classroom dataset.
  - But again as we observed with CT, PhT needs to hold only the list of Physics marks of the students, similarly MaT and ChT need only hold the Maths and Chemistry marks lists.

Example: encapsulation

- What about the average values of the individual subjects?
- Just like CT, we could have objects PhT, MaT and ChT (for Physics, Maths and Chemistry Teachers) that each hold (or have access to) the entire classroom dataset.
  - But again as we observed with CT, PhT needs to hold only the list of Physics marks of the students, similarly MaT and ChT need only hold the Maths and Chemistry marks lists.
- So, let us say that each of these objects are also of the same datatype ClassAve, but their marksList field holds the list of marks of the respective subject.

Example: encapsulation

- What about the average values of the individual subjects?
- Just like CT, we could have objects PhT, MaT and ChT (for Physics, Maths and Chemistry Teachers) that each hold (or have access to) the entire classroom dataset.
  - But again as we observed with CT, PhT needs to hold only the list of Physics marks of the students, similarly MaT and ChT need only hold the Maths and Chemistry marks lists.
- So, let us say that each of these objects are also of the same datatype ClassAve, but their marksList field holds the list of marks of the respective subject.
- Is this enough? What happens when we call PhT.average()?
  - PhT.average() checks if PhT.aValue is -1.
  - If not, it just returns aValue
  - Otherwise, it computes the average from the marks in the marksList - which is just the average of the Physics marks !

Example: encapsulation

- What about the average values of the individual subjects?
- Just like CT, we could have objects PhT, MaT and ChT (for Physics, Maths and Chemistry Teachers) that each hold (or have access to) the entire classroom dataset.
  - But again as we observed with CT, PhT needs to hold only the list of Physics marks of the students, similarly MaT and ChT need only hold the Maths and Chemistry marks lists.
- So, let us say that each of these objects are also of the same datatype ClassAve, but their marksList field holds the list of marks of the respective subject.
- Is this enough? What happens when we call PhT.average()?
  - PhT.average() checks if PhT.aValue is -1.
  - If not, it just returns aValue
  - Otherwise, it computes the average from the marks in the marksList - which is just the average of the Physics marks !
- So the same datatype ClassAve can be used for all the objects, CT, PhT, MaT and ChT !

Compare parametrised procedure with encapsulation

- Procedure Avemarks takes a field as a parameter

Compare parametrised procedure with encapsulation

- Procedure Avemarks takes a field as a parameter
  - We can call Avemarks with Total or the subject name

AveTotal = Avemarks(Total)

AveMaths = Avemarks(Maths)

AvePhysics = Avemarks(Physics)

AveChemistry = Avemarks(Chemistry)

Compare parametrised procedure with encapsulation

- Procedure Avemarks takes a field as a parameter
  - We can call Avemarks with Total or the subject name
- Datatype ClassAve has a procedure average() within it

AveTotal = Avemarks(Total)

AveMaths = Avemarks(Maths)

AvePhysics = Avemarks(Physics)

AveChemistry = Avemarks(Chemistry)

Compare parametrised procedure with encapsulation

- Procedure Avemarks takes a field as a parameter
  - We can call Avemarks with Total or the subject name
- Datatype ClassAve has a procedure average() within it
  - We call average() for each object of ClassAve datatype

AveTotal = Avemarks(Total)

AveMaths = Avemarks(Maths)

AvePhysics = Avemarks(Physics)

AveChemistry = Avemarks(Chemistry)

AveTotal = CT.average()

AveMaths = MaT.average()

AvePhysics = PhT.average()

AveChemistry = ChT.average()

Compare parametrised procedure with encapsulation

- Procedure Avemarks takes a field as a parameter
  - We can call Avemarks with Total or the subject name
- Datatype ClassAve has a procedure average() within it
  - We call average() for each object of ClassAve datatype
  - Advantage: result is stored, next call is much faster
  - Disadvantage: objects need to be created and initialised

AveTotal = Avemarks(Total)

AveMaths = Avemarks(Maths)

AvePhysics = Avemarks(Physics)

AveChemistry = Avemarks(Chemistry)

AveTotal = CT.average()

AveMaths = MaT.average()

AvePhysics = PhT.average()

AveChemistry = ChT.average()

Compare parametrised procedure with encapsulation

- Procedure Avemarks takes a field as a parameter
  - We can call Avemarks with Total or the subject name
- Datatype ClassAve has a procedure average() within it
  - We call average() for each object of ClassAve datatype
  - Advantage: result is stored, next call is much faster
  - Disadvantage: objects need to be created and initialised
- Caller is unaware of what is happening inside

AveTotal = Avemarks(Total)

AveMaths = Avemarks(Maths)

AvePhysics = Avemarks(Physics)

AveChemistry = Avemarks(Chemistry)

AveTotal = CT.average()

AveMaths = MaT.average()

AvePhysics = PhT.average()

AveChemistry = ChT.average()

But what if we the dataset is not static?

- If the dataset changes (consider for example that we add a new student to the class), then the stored average values will be wrong ! How do we deal with this?

But what if we the dataset is not static?

- If the dataset changes (consider for example that we add a new student to the class), then the stored average values will be wrong ! How do we deal with this?
- We need to manage the addition of a new student carefully
  - write a procedure addStudent(newMark) in the ClassAve datatype which takes as parameter the marks of the new student (total, or the respective subject marks)
  - A new student can be added only via the use of this procedure

But what if we the dataset is not static?

- If the dataset changes (consider for example that we add a new student to the class), then the stored average values will be wrong ! How do we deal with this?
- We need to manage the addition of a new student carefully
  - write a procedure addStudent(newMark) in the ClassAve datatype which takes as parameter the marks of the new student (total, or the respective subject marks)
  - A new student can be added only via the use of this procedure
- addStudent will need to append newMark at the end of its marksList

But what if we the dataset is not static?

- If the dataset changes (consider for example that we add a new student to the class), then the stored average values will be wrong ! How do we deal with this?
- We need to manage the addition of a new student carefully
  - write a procedure addStudent(newMark) in the ClassAve datatype which takes as parameter the marks of the new student (total, or the respective subject marks)
  - A new student can be added only via the use of this procedure
- addStudent will need to append newMark at the end of its marksList
- addStudent will also need to reset aValue to -1, so that the average will get computed again

Complete example

- So what does ClassAve datatype look like at the end of all this?

Complete example

- So what does ClassAve datatype look like at the end of all this?
- Has two data fields and two procedures
  - field marksList contains a list of marks (which could be total marks or marks of one subject)
  - field aValue which is either -1 (not computed) or holds the computed average value

Complete example

- So what does ClassAve datatype look like at the end of all this?
- Has two data fields and two procedures
  - field marksList contains a list of marks (which could be total marks or marks of one subject)
  - field aValue which is either -1 (not computed) or holds the computed average value
  - procedure average() returns aValue if it is not -1, else it computes the average of marksList, stores it in aValue and returns it

Complete example

- So what does ClassAve datatype look like at the end of all this?
- Has two data fields and two procedures
  - field marksList contains a list of marks (which could be total marks or marks of one subject)
  - field aValue which is either -1 (not computed) or holds the computed average value
  - procedure average() returns aValue if it is not -1, else it computes the average of marksList, stores it in aValue and returns it
  - procedure addStudent(newMark) appends newMark to the end of marksList and resets aValue to -1

Complete example

- So what does ClassAve datatype look like at the end of all this?
- Has two data fields and two procedures
  - field marksList contains a list of marks (which could be total marks or marks of one subject)
  - field aValue which is either -1 (not computed) or holds the computed average value
  - procedure average() returns aValue if it is not -1, else it computes the average of marksList, stores it in aValue and returns it
  - procedure addStudent(newMark) appends newMark to the end of marksList and resets aValue to -1
- Note that the procedures average and addStudent are assumed to have access to the fields marksList and aValue, so we can reference these fields by just using their name (it is as if they are implicitly passed as parameters to the procedure)

Complete example

- So what does ClassAve datatype look like at the end of all this?
- Has two data fields and two procedures
  - field marksList contains a list of marks (which could be total marks or marks of one subject)
  - field aValue which is either -1 (not computed) or holds the computed average value
  - procedure average() returns aValue if it is not -1, else it computes the average of marksList, stores it in aValue and returns it
  - procedure addStudent(newMark) appends newMark to the end of marksList and resets aValue to -1
- Note that the procedures average and addStudent are assumed to have access to the fields marksList and aValue, so we can reference these fields by just using their name (it is as if they are implicitly passed as parameters to the procedure)
- Note also that the procedures are allowed to (and expected to !) make changes to these fields - so the potential side-effects are made explicit

Complete example

- So what does ClassAve datatype look like at the end of all this?
- Has two data fields and two procedures
  - field marksList contains a list of marks (which could be total marks or marks of one subject)
  - field aValue which is either -1 (not computed) or holds the computed average value
  - procedure average() returns aValue if it is not -1, else it computes the average of marksList, stores it in aValue and returns it
  - procedure addStudent(newMark) appends newMark to the end of marksList and resets aValue to -1
- Note that the procedures average and addStudent are assumed to have access to the fields marksList and aValue, so we can reference these fields by just using their name (it is as if they are implicitly passed as parameters to the procedure)
- Note also that the procedures are allowed to (and expected to !) make changes to these fields - so the potential side-effects are made explicit
- But what if the external world access marksList or aValue directly?

Complete example

- ClassAve should be able to restrict access to its fields

Complete example

- ClassAve should be able to restrict access to its fields
- This is typically done by declaring the field as either a **private** or a **public** field

Complete example

- ClassAve should be able to restrict access to its fields
- This is typically done by declaring the field as either a **private** or a **public** field
- ClassAve has two private fields and two procedures:
  - private field marksList contains a list of marks
  - private field aValue which is either -1 (not computed) or holds the computed average value

Complete example

- ClassAve should be able to restrict access to its fields
- This is typically done by declaring the field as either a **private** or a **public** field
- ClassAve has two private fields and two procedures:
  - private field marksList contains a list of marks
  - private field aValue which is either -1 (not computed) or holds the computed average value
  - public procedure average() returns aValue if it is not -1, else it computes the average of marksList, stores it in aValue and returns it
  - public procedure addStudent(newMark) appends newMark to the end of marksList and resets aValue to -1

Complete example

- ClassAve should be able to restrict access to its fields
- This is typically done by declaring the field as either a **private** or a **public** field
- ClassAve has two private fields and two procedures:
  - private field marksList contains a list of marks
  - private field aValue which is either -1 (not computed) or holds the computed average value
  - public procedure average() returns aValue if it is not -1, else it computes the average of marksList, stores it in aValue and returns it
  - public procedure addStudent(newMark) appends newMark to the end of marksList and resets aValue to -1
- Note that a procedure could also be declared private in which case it is not available to the outside world

Example: Shopping Bills dataset

- Questions asked of the shopping bill dataset
  - How many items of a specific category (e.g. shirts) were sold?
  - What is the minimum, maximum and average unit price for that category?

Example: Shopping Bills dataset

- Questions asked of the shopping bill dataset
  - How many items of a specific category (e.g. shirts) were sold?
  - What is the minimum, maximum and average unit price for that category?
- We could define a datatype Category and create an object Shirts of this datatype

Example: Shopping Bills dataset

- Questions asked of the shopping bill dataset
  - How many items of a specific category (e.g. shirts) were sold?
  - What is the minimum, maximum and average unit price for that category?
- We could define a datatype Category and create an object Shirts of this datatype
- What are its fields?

Example: Shopping Bills dataset

- Questions asked of the shopping bill dataset
  - How many items of a specific category (e.g. shirts) were sold?
  - What is the minimum, maximum and average unit price for that category?
- We could define a datatype Category and create an object Shirts of this datatype
- What are its fields?
  - copy of the entire shopping bill dataset can be stored, but this is wasteful

Example: Shopping Bills dataset

- Questions asked of the shopping bill dataset
  - How many items of a specific category (e.g. shirts) were sold?
  - What is the minimum, maximum and average unit price for that category?
- We could define a datatype Category and create an object Shirts of this datatype
- What are its fields?
  - copy of the entire shopping bill dataset can be stored, but this is wasteful
  - We only need the list of items (i.e. rows of the bills) which are of that category
  - Category could have a private field itemList which carries all the row items from all the bills which are all of the same category

Example: Shopping Bills dataset

- Questions asked of the shopping bill dataset
  - How many items of a specific category (e.g. shirts) were sold?
  - What is the minimum, maximum and average unit price for that category?
- We could define a datatype Category and create an object Shirts of this datatype
- What are its fields?
  - copy of the entire shopping bill dataset can be stored, but this is wasteful
  - We only need the list of items (i.e. rows of the bills) which are of that category
  - Category could have a private field itemList which carries all the row items from all the bills which are all of the same category
- The datatype could also encapsulate procedures that provide the desired information:
  - count() returns the number of items (i.e. size of the list)
  - min() returns the least value of unit price at which the category item was sold
  - max() returns the largest value of the unit price
  - average() returns average across all unit prices for that category

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call
- What if we want to do this for a particular shop? Or for a specific customer?
  - The itemList can store only the items sold by that shop
  - ... or by a specific customer
  - The procedures will all work without any change !

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call
- What if we want to do this for a particular shop? Or for a specific customer?
  - The itemList can store only the items sold by that shop
  - ... or by a specific customer
  - The procedures will all work without any change !
- Multiple objects of type Category can be created - the list being filtered by category alone, or by a combination of category and customer name and shop name, or anything else.

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call
- What if we want to do this for a particular shop? Or for a specific customer?
  - The itemList can store only the items sold by that shop
  - ... or by a specific customer
  - The procedures will all work without any change !
- Multiple objects of type Category can be created - the list being filtered by category alone, or by a combination of category and customer name and shop name, or anything else.
- What if we want to accelerate the execution of these procedures?

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call
- What if we want to do this for a particular shop? Or for a specific customer?
  - The itemList can store only the items sold by that shop
  - ... or by a specific customer
  - The procedures will all work without any change !
- Multiple objects of type Category can be created - the list being filtered by category alone, or by a combination of category and customer name and shop name, or anything else.
- What if we want to accelerate the execution of these procedures?
  - Just store their results in private fields - say cValue for count, minValue for min, maxValue for max and aValue for average.

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call
- What if we want to do this for a particular shop? Or for a specific customer?
  - The itemList can store only the items sold by that shop
  - ... or by a specific customer
  - The procedures will all work without any change !
- Multiple objects of type Category can be created - the list being filtered by category alone, or by a combination of category and customer name and shop name, or anything else.
- What if we want to accelerate the execution of these procedures?
  - Just store their results in private fields - say cValue for count, minValue for min, maxValue for max and aValue for average.
- Can you see how the pattern for encapsulating the Category is quite similar to that for encapsulating the ClassAve?

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call
- If Shirts is an object of this datatype, we may want to find out how many shirts were sold
  - Note that this may not be the same as count(), since one row in the bill can have multiple items

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call
- If Shirts is an object of this datatype, we may want to find out how many shirts were sold
  - Note that this may not be the same as count(), since one row in the bill can have multiple items
  - We can write a procedure called number() which finds the sum of the quantity field of the list items

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call
- If Shirts is an object of this datatype, we may want to find out how many shirts were sold
  - Note that this may not be the same as count(), since one row in the bill can have multiple items
  - We can write a procedure called number() which finds the sum of the quantity field of the list items
- But number() makes sense only if the category is sold in discrete units (like shirts). What if the category quantity is measured in Kg (for e.g. grapes)?

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call
- If Shirts is an object of this datatype, we may want to find out how many shirts were sold
  - Note that this may not be the same as count(), since one row in the bill can have multiple items
  - We can write a procedure called number() which finds the sum of the quantity field of the list items
- But number() makes sense only if the category is sold in discrete units (like shirts). What if the category quantity is measured in Kg (for e.g. grapes)?
  - For such categories, we can define a procedure quantity() that finds the sum of the (possibly fractional) quantity fields

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
  - private field itemList that carries the list of items, which is hidden
  - public procedures count(), min(), max() and average() that anyone can call
- If Shirts is an object of this datatype, we may want to find out how many shirts were sold
  - Note that this may not be the same as count(), since one row in the bill can have multiple items
  - We can write a procedure called number() which finds the sum of the quantity field of the list items
- But number() makes sense only if the category is sold in discrete units (like shirts). What if the category quantity is measured in Kg (for e.g. grapes)?
  - For such categories, we can define a procedure quantity() that finds the sum of the (possibly fractional) quantity fields
- The issue is that just like number() does not make sense for grapes, a measure of quantity in kg does not make sense for shirts !

Using derived datatypes

- While the generic procedures count(), min(), max() and average() worked for all the categories, there could be procedures that work only for some categories and don't work for others
  - number() works for shirts but not for grapes
  - quantity() works for grapes, but may not work for shirts

Using derived datatypes

- While the generic procedures count(), min(), max() and average() worked for all the categories, there could be procedures that work only for some categories and don't work for others
  - number() works for shirts but not for grapes
  - quantity() works for grapes, but may not work for shirts
- We can write derived datatypes to deal with this

Using derived datatypes

- While the generic procedures count(), min(), max() and average() worked for all the categories, there could be procedures that work only for some categories and don't work for others
  - number() works for shirts but not for grapes
  - quantity() works for grapes, but may not work for shirts
- We can write derived datatypes to deal with this
  - Create derived types NumberCategory and QuantityCategory of Category

Using derived datatypes

- While the generic procedures count(), min(), max() and average() worked for all the categories, there could be procedures that work only for some categories and don't work for others
  - number() works for shirts but not for grapes
  - quantity() works for grapes, but may not work for shirts
- We can write derived datatypes to deal with this
  - Create derived types NumberCategory and QuantityCategory of Category
  - All the generic procedures count(), min(), max() and average() are available for objects of these derived types also

Using derived datatypes

- While the generic procedures count(), min(), max() and average() worked for all the categories, there could be procedures that work only for some categories and don't work for others
  - number() works for shirts but not for grapes
  - quantity() works for grapes, but may not work for shirts
- We can write derived datatypes to deal with this
  - Create derived types NumberCategory and QuantityCategory of Category
  - All the generic procedures count(), min(), max() and average() are available for objects of these derived types also
- We could then add more specific procedures to each derived type that will be available only to objects of that derived type
  - number() procedure is defined in NumberCategory
  - quantity() procedure is defined in QuantityCategory

Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate

Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate
  - This is achieved by allowing procedures in addition to fields in a datatype

Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate
  - This is achieved by allowing procedures in addition to fields in a datatype
  - Procedures are the interfaces for others to interact with the objects of this datatype

Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate
  - This is achieved by allowing procedures in addition to fields in a datatype
  - Procedures are the interfaces for others to interact with the objects of this datatype
  - The procedures can have side effects in terms of modifying the field values of the object

Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate
  - This is achieved by allowing procedures in addition to fields in a datatype
  - Procedures are the interfaces for others to interact with the objects of this datatype
  - The procedures can have side effects in terms of modifying the field values of the object
  - Fields (or procedures) can be hidden from the outside world by marking them as private

Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate
  - This is achieved by allowing procedures in addition to fields in a datatype
  - Procedures are the interfaces for others to interact with the objects of this datatype
  - The procedures can have side effects in terms of modifying the field values of the object
  - Fields (or procedures) can be hidden from the outside world by marking them as private
- Encapsulation provides an increased level of modularisation when compared to procedures

Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate
  - This is achieved by allowing procedures in addition to fields in a datatype
  - Procedures are the interfaces for others to interact with the objects of this datatype
  - The procedures can have side effects in terms of modifying the field values of the object
  - Fields (or procedures) can be hidden from the outside world by marking them as private
- Encapsulation provides an increased level of modularisation when compared to procedures
  - Allows state to be retained between calls ... can be used to speed up the procedures

Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate
  - This is achieved by allowing procedures in addition to fields in a datatype
  - Procedures are the interfaces for others to interact with the objects of this datatype
  - The procedures can have side effects in terms of modifying the field values of the object
  - Fields (or procedures) can be hidden from the outside world by marking them as private
- Encapsulation provides an increased level of modularisation when compared to procedures
  - Allows state to be retained between calls ... can be used to speed up the procedures
  - Side effects are made explicit and more natural

Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate
  - This is achieved by allowing procedures in addition to fields in a datatype
  - Procedures are the interfaces for others to interact with the objects of this datatype
  - The procedures can have side effects in terms of modifying the field values of the object
  - Fields (or procedures) can be hidden from the outside world by marking them as private
- Encapsulation provides an increased level of modularisation when compared to procedures
  - Allows state to be retained between calls ... can be used to speed up the procedures
  - Side effects are made explicit and more natural
  - We can use derived types to extend the functionality to specific instances when needed

Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate
  - This is achieved by allowing procedures in addition to fields in a datatype
  - Procedures are the interfaces for others to interact with the objects of this datatype
  - The procedures can have side effects in terms of modifying the field values of the object
  - Fields (or procedures) can be hidden from the outside world by marking them as private
- Encapsulation provides an increased level of modularisation when compared to procedures
  - Allows state to be retained between calls ... can be used to speed up the procedures
  - Side effects are made explicit and more natural
  - We can use derived types to extend the functionality to specific instances when needed
  - But the disadvantage is that the objects need to be created and initialised at the start

Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate
  - This is achieved by allowing procedures in addition to fields in a datatype
  - Procedures are the interfaces for others to interact with the objects of this datatype
  - The procedures can have side effects in terms of modifying the field values of the object
  - Fields (or procedures) can be hidden from the outside world by marking them as private
- Encapsulation provides an increased level of modularisation when compared to procedures
  - Allows state to be retained between calls ... can be used to speed up the procedures
  - Side effects are made explicit and more natural
  - We can use derived types to extend the functionality to specific instances when needed
  - But the disadvantage is that the objects need to be created and initialised at the start
- We can find common "object oriented computing" patterns between different examples