CSC108: Introduction to Computer Programming

Lecture 9

Wael Aboulsaadat

Acknowledgment: these slides are based on material by: Velian Pandeliev, Diane Horton, Michael Samozi, Jennifer Campbell, and Paul Gries from CS UoT

Searching

Linear Search

```
def linear_search(lst,item):



if __name__ == "__main__":
    print linear_search([9,1,5,7,8,3,4,6],6)
```

Binary Search

```
def binary_search(lst,item):




if __name__ == "__main__":
    print binary_search([9,1,5,7,8,3,4,6],6)
```

Efficiency

How do we judge if an algorithm is written efficiently ?

Introduction to Efficiency

- Possible measures of efficiency:
  - development time (code & test)
  - program size
  - run time*
  - memory usage*
  - bandwidth
- In general, “efficiency” taken to mean time & space
- What’s wrong with just running the code (stopwatch approach)?
  - influenced by hardware
  - influenced by system software
  - influenced by other activity
  - influenced by data selection
- Better to just analyse code, independent of these factors

How is Efficiency Measured?

- Actions of interest:
  - Comparisons (read a memory value)
  - Assignments (setting a memory value)

- Why?
  - memory operations involve extra overhead
    - Fastest to slowest: CPU, memory, hard drive, external
  - memory operations are performed repeatedly

How is Efficiency Measured?

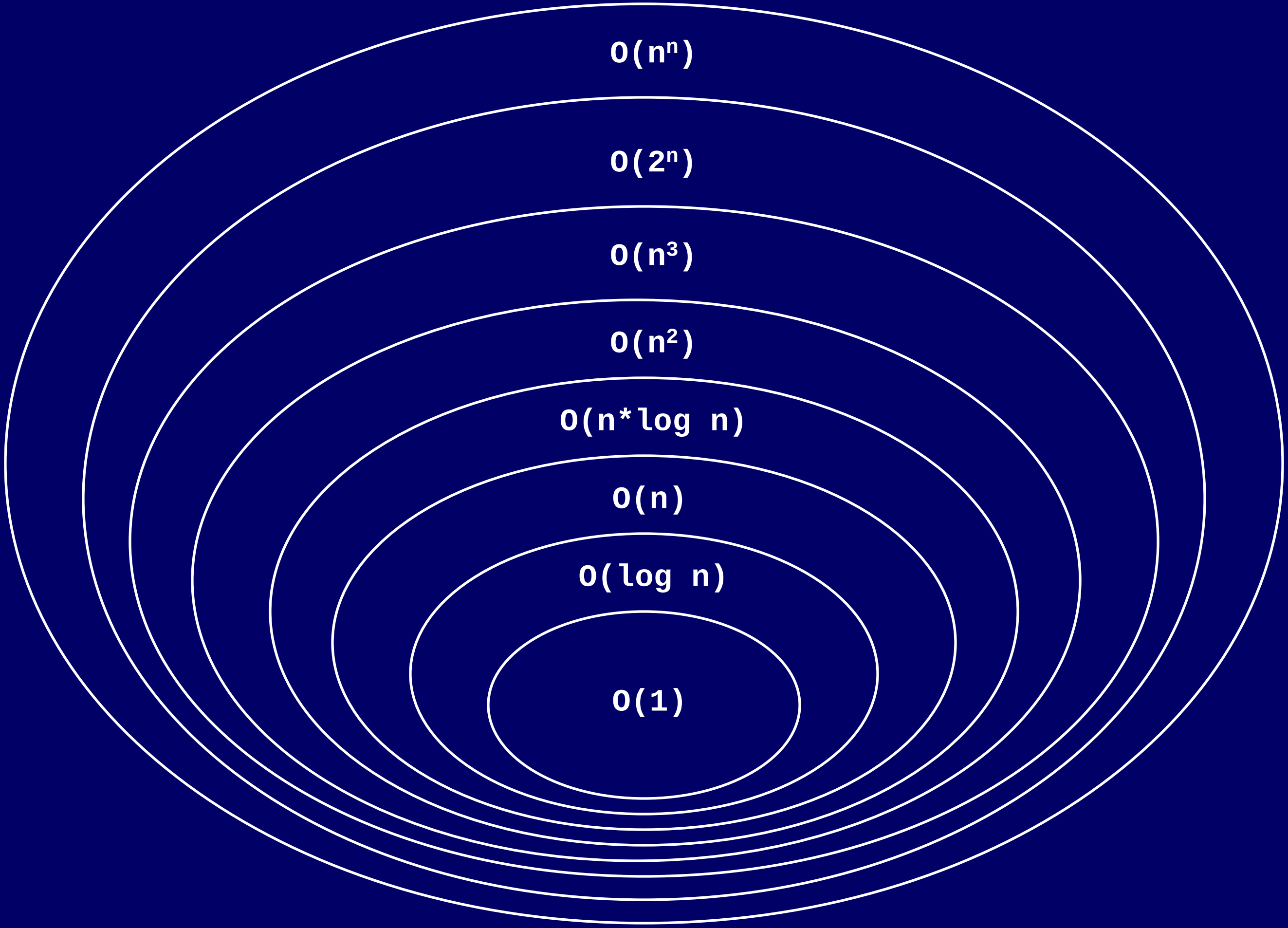
- Interesting behavior:
  - worst-case analysis (worst input value and/or structure)
  - best-case analysis (best input value and/or structure)
  - average-case analysis (complicated)

- Uninteresting behavior:
  - Trivial case

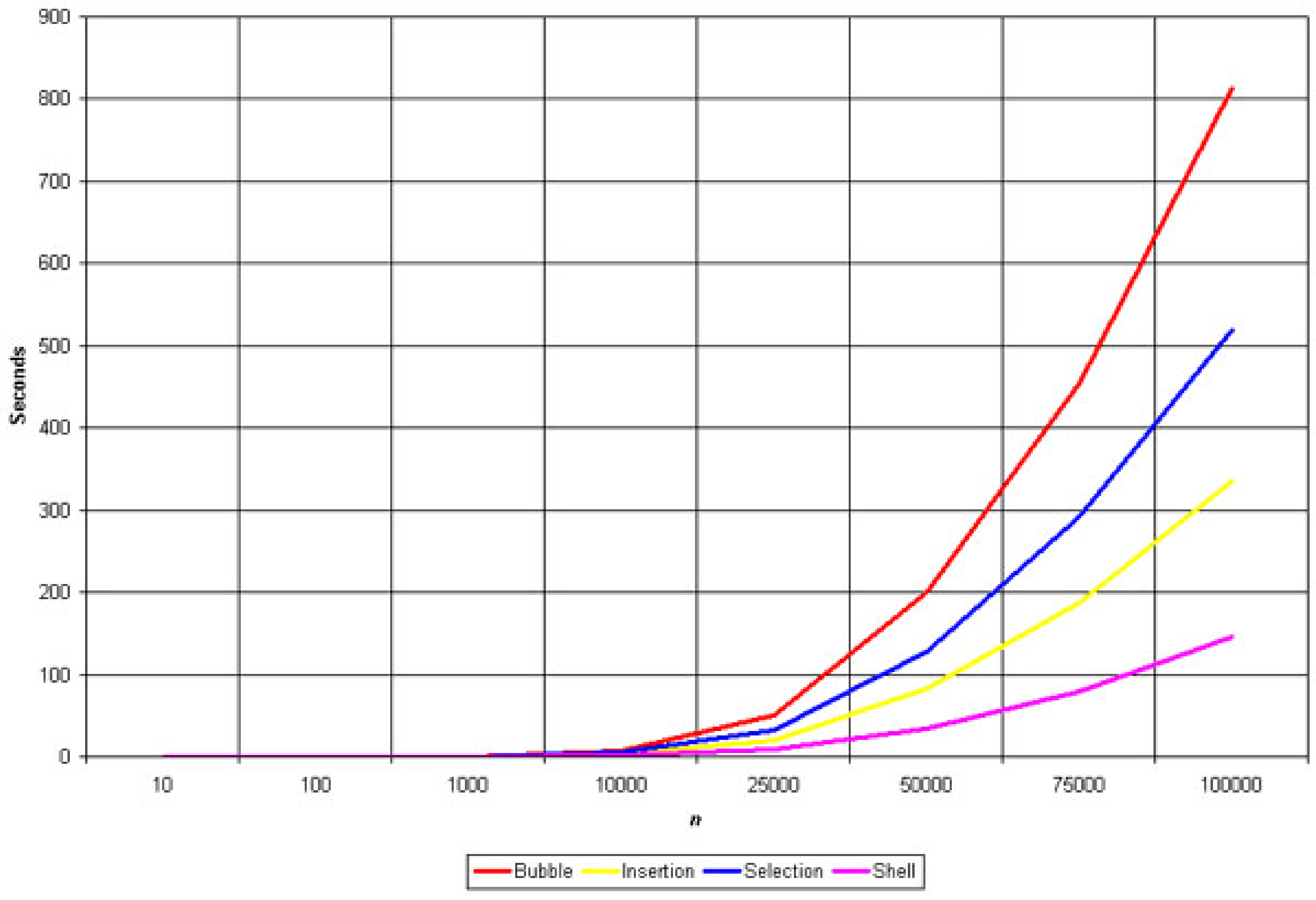
Asymptotic Analysis: Upper Bound

- Growth rate = rate at which an algorithm’s cost grows as its input grows

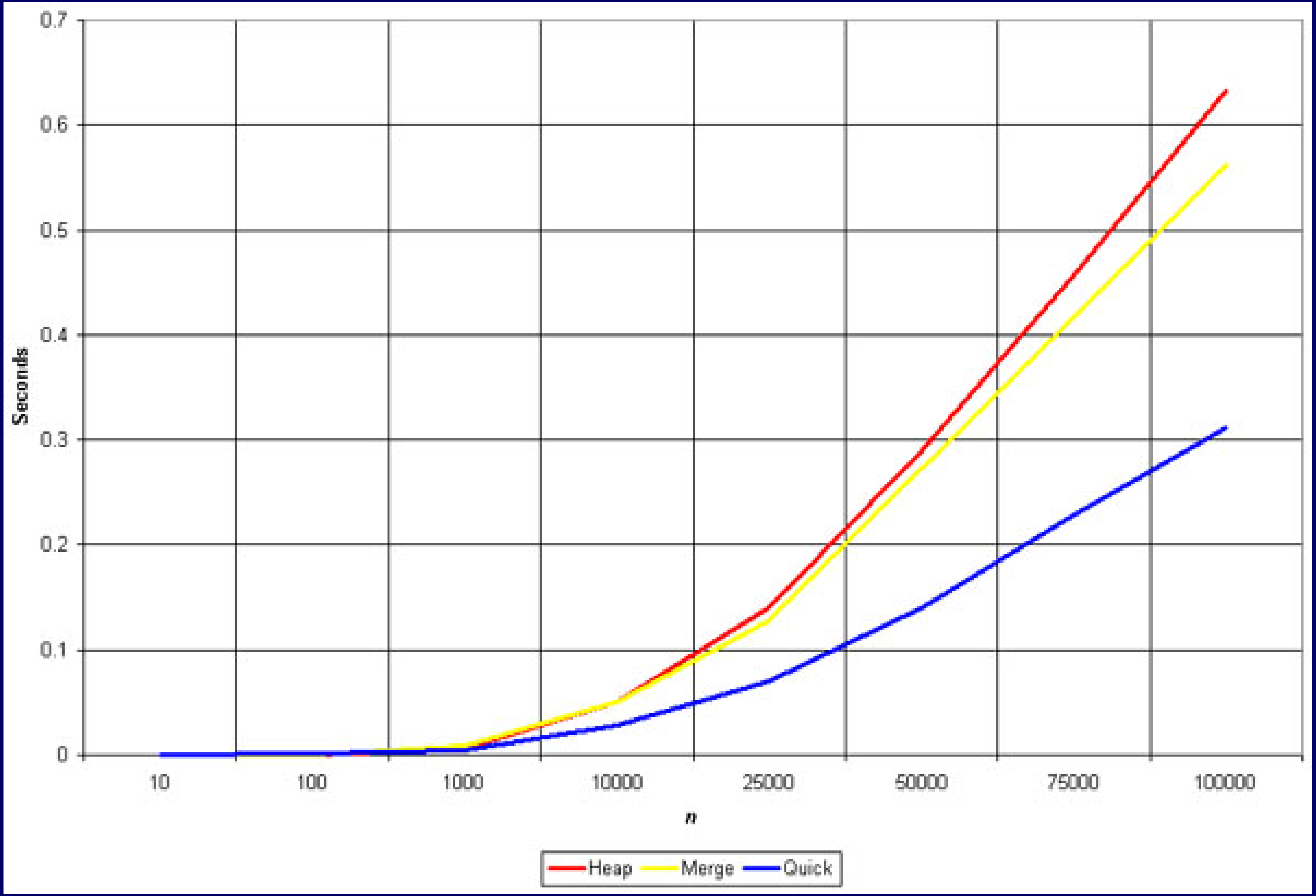
- Algorithm analysis concerns itself with the number of “basic operations” required to process input of a certain size
  - “basic operations” are usually memory accesses


O(n^n)
O(2^n)
O(n^3)
O(n^2)
O(n*log n)
O(n)
O(log n)
O(1)

O(n^2) Sorts


900
800
700
600
500
400
300
200
100
0
Seconds
10
100
1000
10000
25000
50000
75000
100000
n
Bubble
Insertion
Selection
Shell

O(n log n) Sorts



id & copy

id function

- Recall: variable name is just for us – each variable value is stored in a memory cell.
- Each memory cell has an address.
- E.g.

Copying Variables

- For immutable objects, 'copying' really involves two separate variables referring to the same object. When one is changed, it does not affect the other since it's simply referring to a new value.
- For mutable objects, this isn't as easy:

```
>>> x = [1, 2, 3]
>>> y = x
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
[1, 2, 3, 4]
```

Copying Instances of a Class

- Copying a mutable object, even a simple one, involves allocating a new space in memory and creating new references to the object's components.
- For lists, this can be done using slice notation:

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> id(x)
4603616
>>> id(y)
4627672
>>> x.append(4)
>>> y
[1, 2, 3]
```

The copy Module

- Python has a built-in module called copy that can copy arbitrary objects.

```
>>> x = [1, 2, 3]
>>> id(x)
4603616
>>> import copy
>>> y = copy.copy(x)
>>> id(y)
4627672
>>> x.append(4)
>>> y
[1, 2, 3]
```

The copy() Function

- copy.copy() works on user-defined classes too.

- Using a class – Point – that represents a point with x,y:

```
>>> import copy
>>> a = Point(1,3)
>>> id(a)
4603616
>>> b = a
>>> id(b)
4603616
>>> b = copy.copy(a)
>>> id(b)
4899521
```

The copy() Function

- Consider the class Creature:

```
class Creature():
        def __init__(self,n,limblist):
                self.name = n
                self.limbs = limblist
>>> import copy
>>> g = Creature("Galgarag", ["wing","wing","claw","tail"])
>>> id(g.limbs)
4356664
>>> g2 = copy.copy(g)
>>> id(g2.limbs)
4356664
```

The copy() Function

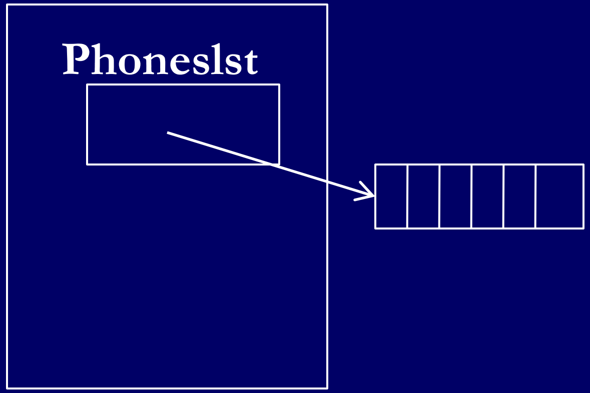
- While copy() creates a new copy of the instance of a class, it does not create new copies of its attributes. Instead, it creates references to them.

- Changing a copy's immutable attributes will still not affect the original. However, for mutable attributes, the original and the copy are still referring to the same actual object, and changing it for one will change it for the other.

- copy() is a method that creates a **shallow copy** of an object: a copy containing only references to its attributes.

Shallow Copy

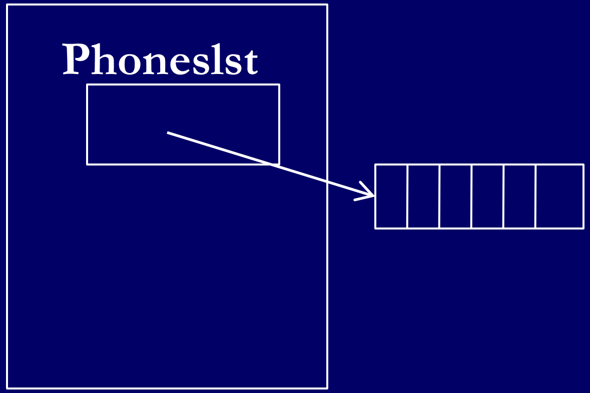
X = Person()



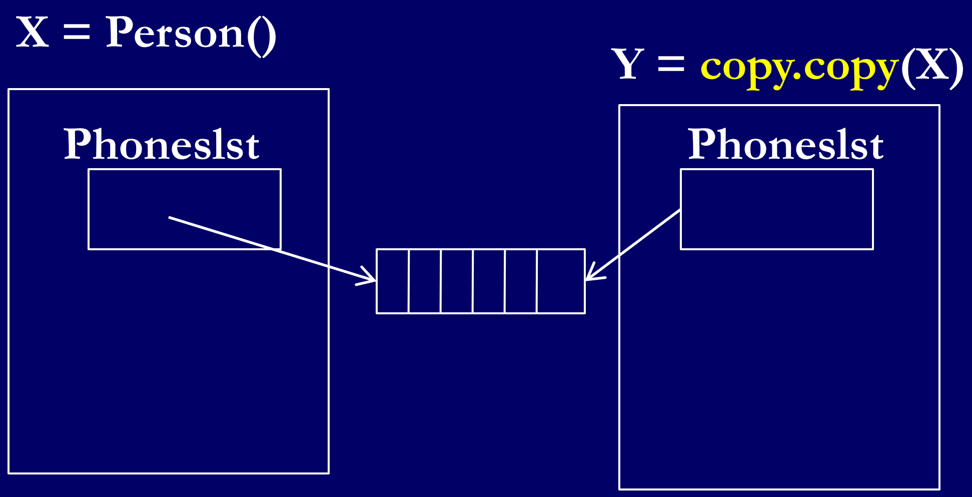
Shallow Copy

X = Person()

Y = copy.copy(X)



Shallow Copy



X = Person()

Y = copy.copy(X)

Phoneslst

Phoneslst

The deepcopy() Function

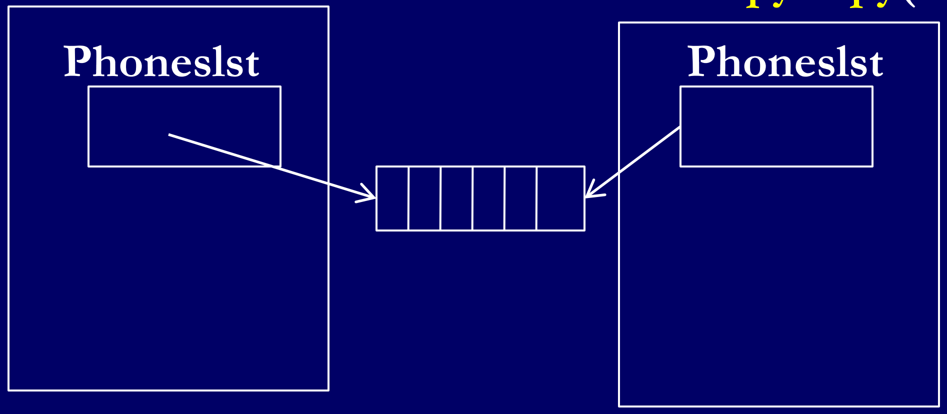
- The copy module has the method deepcopy() that makes a **deep copy of an object (which** includes copying its attributes):

```
>>> import copy
>>> g = Creature("Galgarag",
                            ["wing","wing","claw","tail"])
>>> id(g.limbs)
4356664
>>> g2 = copy.deepcopy(g)
>>> id(g2.limbs)
4354344
```

Shallow Copy vs. Deep Copy

X = Person()

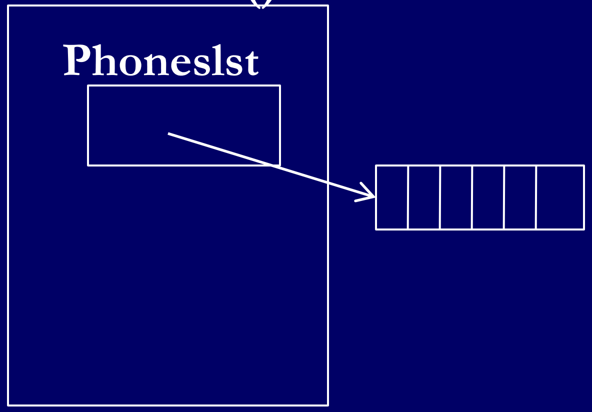
Y = copy.copy(X)



Phoneslst

Phoneslst

X = Person()

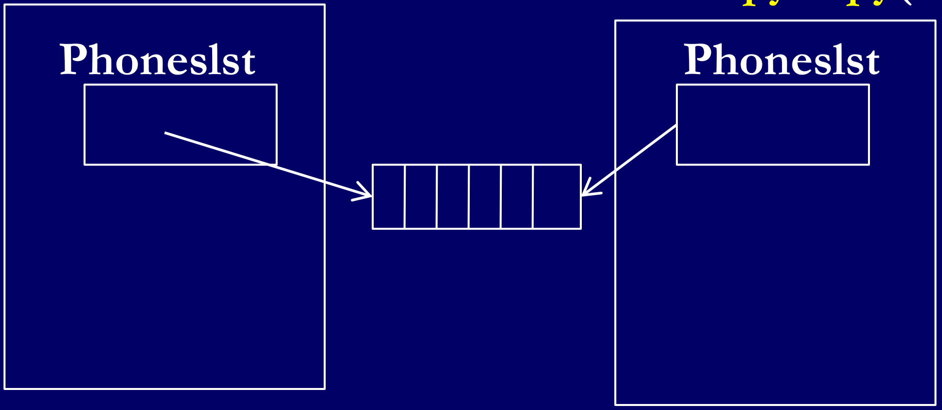


Phoneslst

Shallow Copy vs. Deep Copy

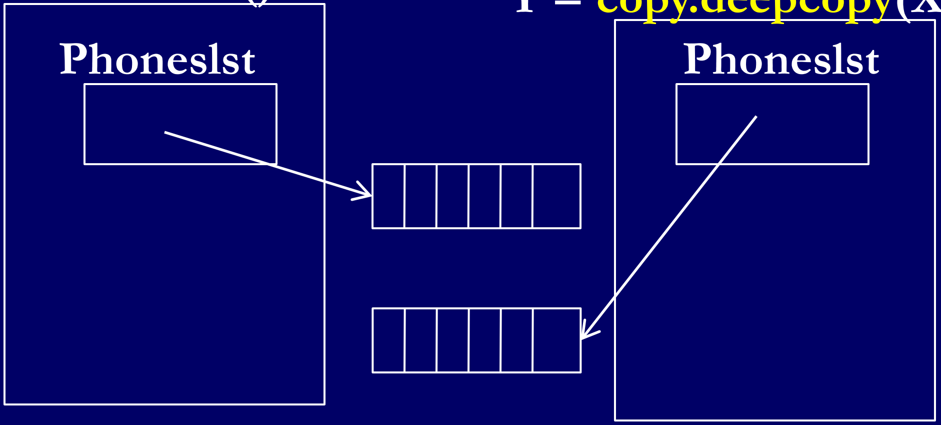
X = Person()

Y = copy.copy(X)



X = Person()

Y = copy.deepcopy(X)



The deepcopy() Function

- deepcopy() will not only copy the object and its attributes, but also its attributes' attributes, as deep as it needs to go.

- Consider the following:

```
class Body():
  def __init__(self):
        self.head = Head(self)

class Head():
  def __init__(self, b):
        self.body = b
```

The deepcopy() Function

- If deepcopy() was not implemented carefully, and it had tried to copy mutually referring objects, it would have run forever.
- Thankfully, deepcopy() is aware of this and will not enter infinite loops of the sort.

Classes & Objects (revisited)

System Methods

- __init__ is an example of system methods
- We will see a few more system methods that are used with user-defined classes.
- They are all flanked by two underscores and include:

```
def __str__ ():
'''returns a string representation of the object. '''

def __eq__ ():
'''returns whether two objects of a class are equal. '''

def  __cmp__ ():
'''which determines how objects compare to each other. '''
```

__str__

- By default:

```
>>> print apoint
<__main__.Point instance at 0x45d800>
```

- __str__ returns the string representation of the object, i.e. what str(object) should return and how the object is printed.

```
def __str__(self):
'''Return a string to represent a Point object'''
        return '(%d, %d)' % (self.x, self.y)
```

- After defining __str__:

```
>>> print apoint
(3, 5)
```

__eq__

- __eq__ returns True if two objects are equal, however we choose to define equality. It determines what obj1 == obj2 should return.
- If __eq__ is not defined, obj1 == obj2 will return True iff obj1 and obj2 are referring to the same object (i.e. the same memory address):

```
>>> x = Point(3,5)
>>> y = Point(3,5)
>>> x == y
False
>>> id(x) == id(y)
False
```

__eq__

- __eq__ allows us to specify that two objects do not need to be the same object to be equal. With Point objects, perhaps we want them to be equal if they refer to the same point:

```
def __eq__(self, other):
'''Return True iff self == other'''
        return self.x == other.x  and self.y == other.y
```

- Now:

```
>>> x = Point(3,5)
>>> y = Point(3,5)
>>> z = Point(3,4)
>>> x == y
True
```

```
>>> x == z
False
```

Comparisons methods

- __eq__ is part of a suite of methods which determine the action of all comparison operators:

```
__lt__(self, other)        # <
__le__(self, other)        # <=
__eq__(self, other)        # ==
__ne__(self, other)        # !=
__gt__(self, other)        # >
__ge__(self, other)        # >=
```

__cmp__

- __cmp__ is a method that can be used if the other comparison methods aren't defined. The result of the __cmp__ method determines the relationship between two objects of the class:

```
def __cmp__(self, other):
```

- It should return a negative number if self is less than other, 0 if they're equal and a positive number if self is greater than other.
- Note that since __cmp__ includes an equality condition, if we include __cmp__ we don't need to specifically include __eq__.

__getitem__

- __getitem__ enables the programmer to use [ ] with a custom class

- Does not make sense unless that class has a list of items inside it.

```
class Building:
    def __getitem__ (self, index):

>> mybuilding = Building()
>> mybuilding[1]
```

__contains__

- __contains__ enables the programmer to use the membership in operator with a custom class

- Does not make sense unless that class has a list of items inside it.

```
class Building:
    def __contains__ (self, item):

>> mybuilding = Building()
>> john          = Person()
>> john  in mybuilding
```

__len__

- __len__ enables the programmer to use the function len with custom class

- Does not make sense unless that class has some kind of length attribute:

```
class Street:
        def __len__ (self):

>> college_street = Street()
>> length(college_street)
```

__add__

- __add__ enables the programmer to add one object to another in an expression !
- Does not make sense unless the operation has a meaning for the class context!

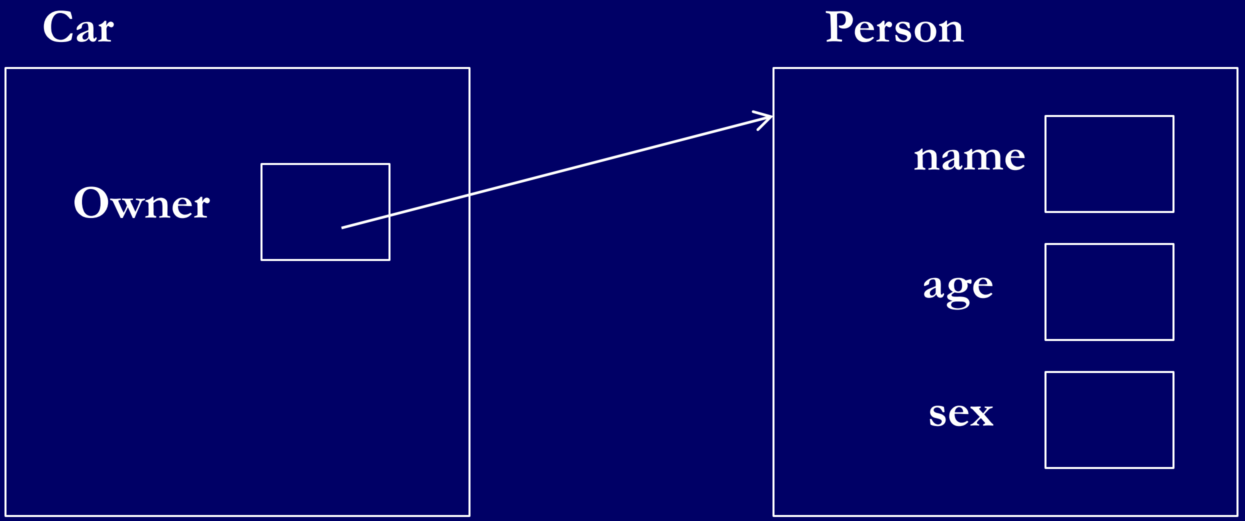
Relationships Between Classes

- As the building blocks of more complex systems, objects can be designed to interact with each other in one of three ways:
- **Association: an object is aware of another object** and holds a reference to it
- **Composition: objects combining to create more** complex ones
- **Inheritance: objects are created as extensions of** other objects with additional properties

Association

- In an associative **uses relationship, an object is** aware of another complex object and can communicate with it.

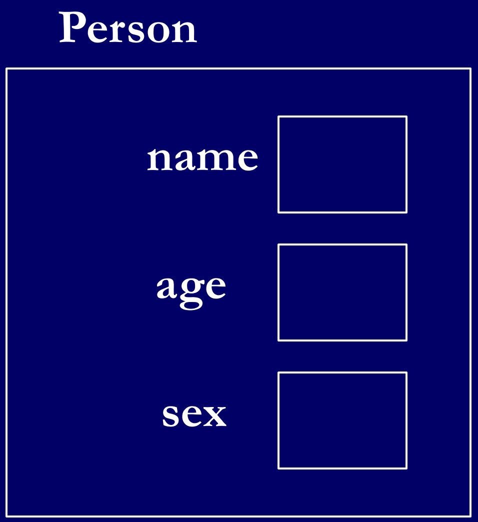
- Example: a Car has an owner attribute which is a Person.



Composition

- In a compositional **has-a relationship**, **an object is** made up of less complex objects.

- Examples:
  - A Person has name, age and sex.
  - A Movie object is composed of string objects title and genre and integer object year.



Public and Private Data: atom class

```
class atom:
  def __init__(self,atno,x,y,z):
        self.atno = atno
        self.position = (x,y,z)
```

Public and Private Data

- Uptill now all attributes (data) in a class is public, thus we, my mistake, could do something really stupid like

```
>>> at = atom(6,0.,0.,0.)
>>> at.position = 'Grape Jelly'
```

that would break any function that used at.position

Public and Private Data

- We therefore need to protect the at.position and provide accessors to this data
  - Encapsulation or Data Hiding
  - accessors are "gettors" and "settors"

- Encapsulation is particularly important when other developers use your class

Public and Private Data

- In Python anything with two leading underscores is private

    __a, __my_variable

Encapsulated Atom

```
class atom:
  def __init__(self,atno,x,y,z):
        self.atno = atno
        self.__position = (x,y,z)      #position is private
  def getposition(self):
        return self.__position
  def setposition(self,x,y,z):
        self.__position = (x,y,z)
  def translate(self,x,y,z):
        x0,y0,z0 = self.__position
        self.__position = (x0+x,y0+y,z0+z)
```

Why Encapsulate?

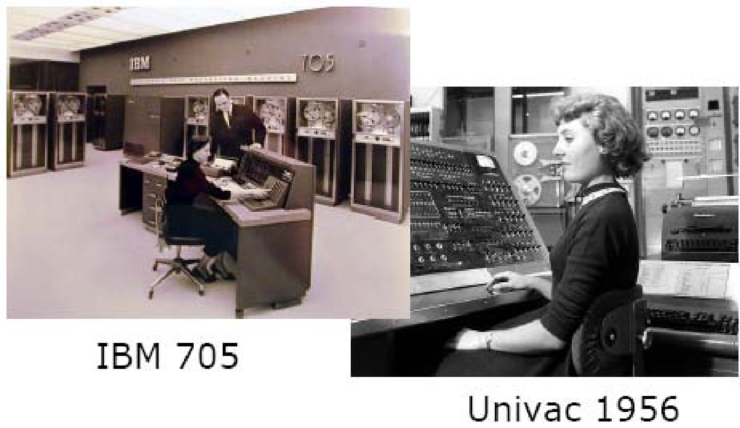
- By defining a specific interface you can keep other modules from doing anything incorrect to your data

- By limiting the functions you are going to support, you leave yourself free to change the internal data without messing up your users
  - Makes code more modular, since you can change large parts of your classes without affecting other parts of the program, so long as they only use your public functions

GUI

Typical command line program

- Non-interactive
- Linear execution



IBM 705

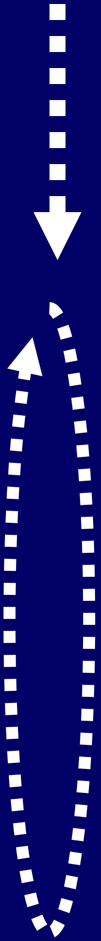
Univac 1956

program:

```
main()
{
	code;
	code;
	code;
	code;
	code;
	code;
	code;
	code;
	code;
	code;
	code;
	code;
}
```

Interactive command line program

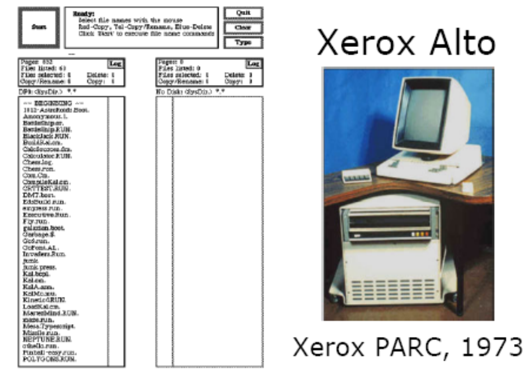
- User input commands
- Non-linear execution
- Unpredictable order
- Much idle time

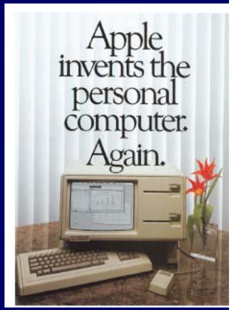


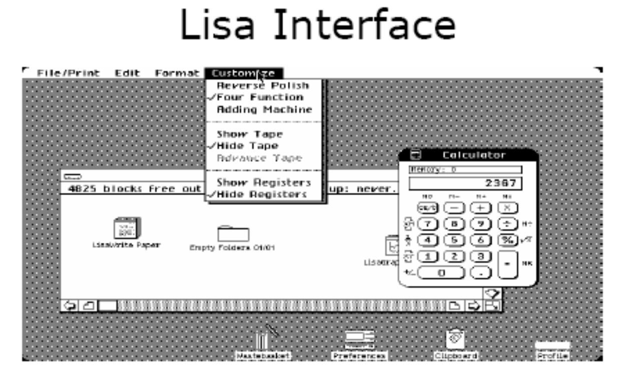
program:

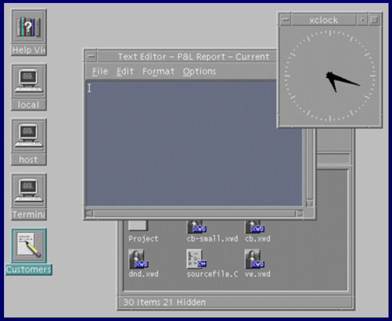
```
while True:
      cmd=getCommand()
       if  cmd == 1:
```

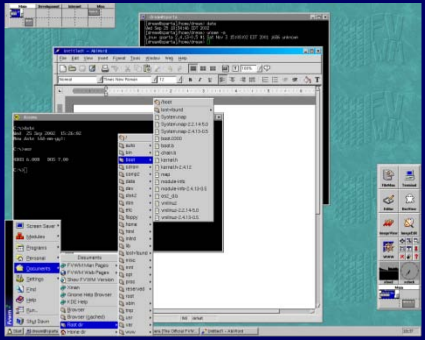
Interactive Graphical User Interface







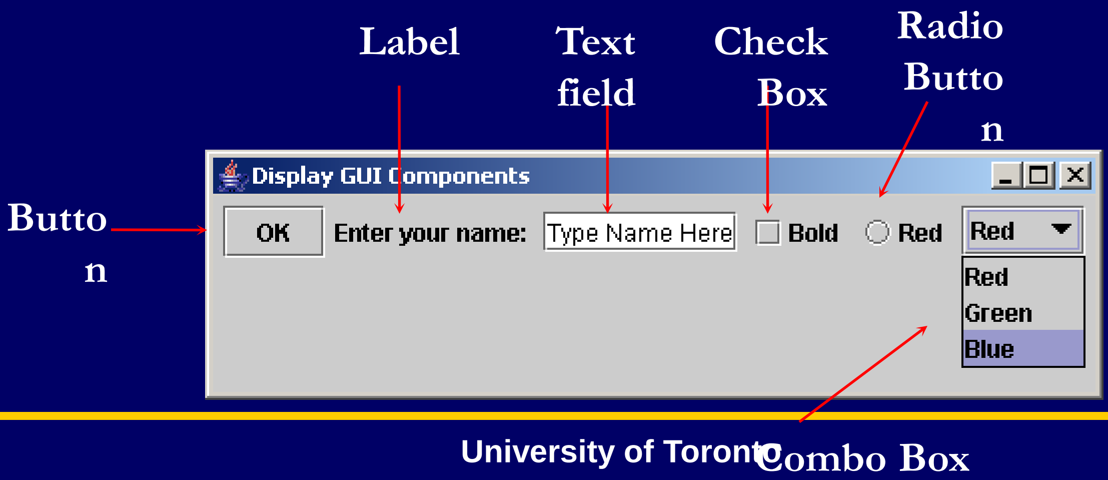




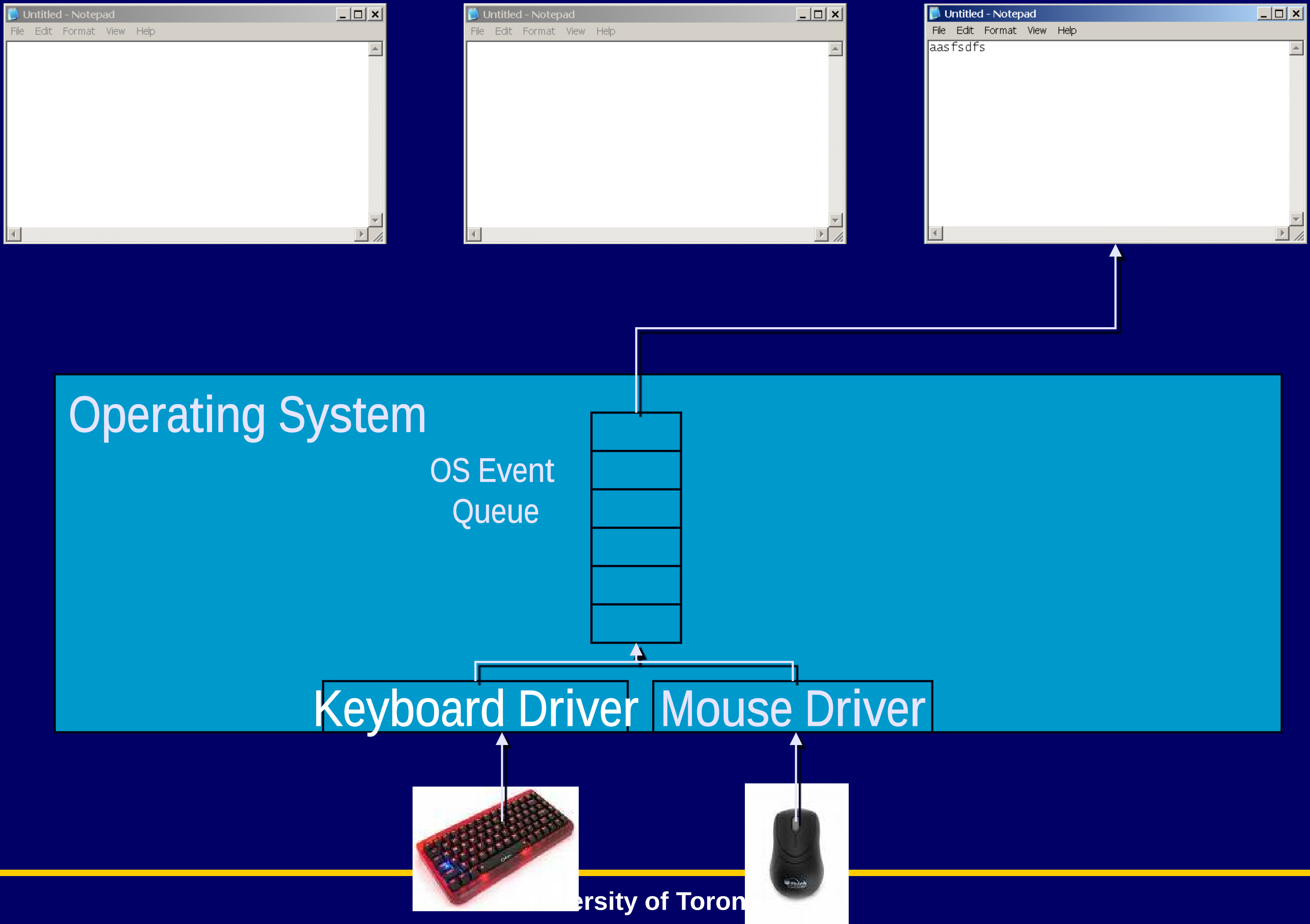
Virtual PC (Red Hat Linux 6.1)
Netscape: REDHAT.COM -- Serving the Linux and Open Source Communities
redhat.com
support
Search
Red Hat Store
Support & Downloads

Interactive Graphical User Interface

- What's make a GUI GUI?
  - Windows
  - Selection controls: drop-downs, radio-buttons, check boxes, menus,..
  - Activation controls: buttons, icons
  - Input controls: text fields, text areas
  - Structure information visually: lists, grids, trees, labels



Input Events



Untitled - Notepad
File Edit Format View Help
Untitled - Notepad
File Edit Format View Help
Untitled - Notepad
File Edit Format View Help
aasfsdfs

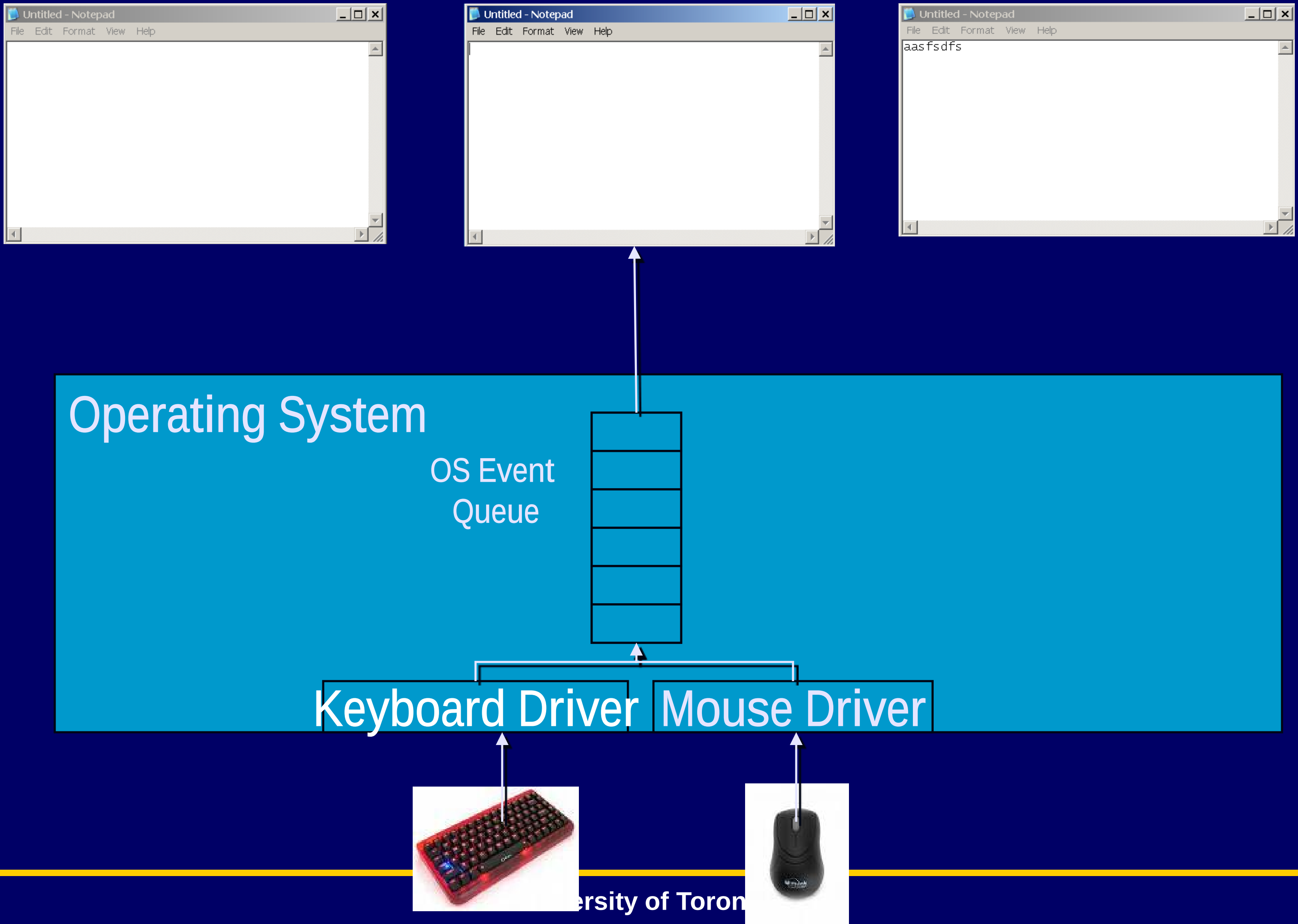
Operating System

OS Event Queue

Keyboard Driver

Mouse Driver

Input Events


Untitled - Notepad
File Edit Format View Help
Untitled - Notepad
File Edit Format View Help
Untitled - Notepad
File Edit Format View Help
aasfsdfs

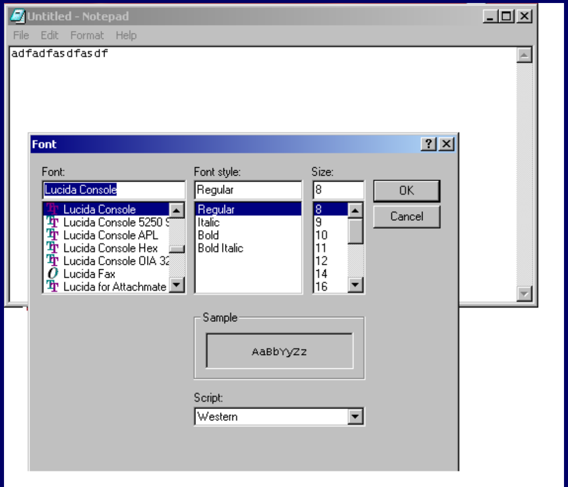
Operating System

OS Event Queue

Keyboard Driver

Mouse Driver

Input Events



Untitled - Notepad

File Edit Format Help

```
adfadfasdfasdf
```

Font

Font: Lucida Console

- Lucida Console
- Lucida Console 5250 S
- Lucida Console APL
- Lucida Console Hex
- Lucida Console OIA 32
- Lucida Fax
- Lucida for Attachmate

Font style: Regular

- Regular
- Italic
- Bold
- Bold Italic

Size: 8

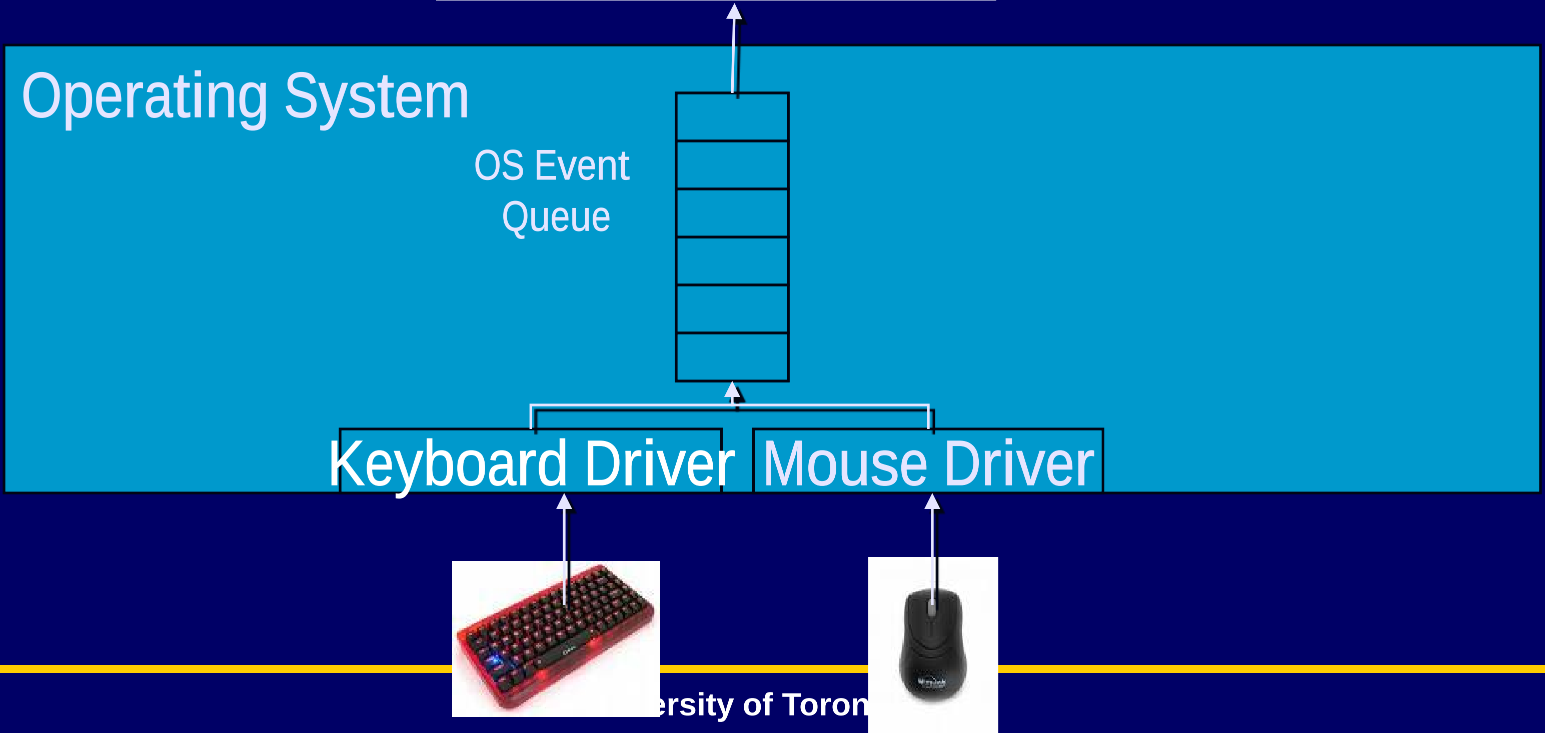
- 8
- 9
- 10
- 11
- 12
- 14
- 16

OK

Cancel

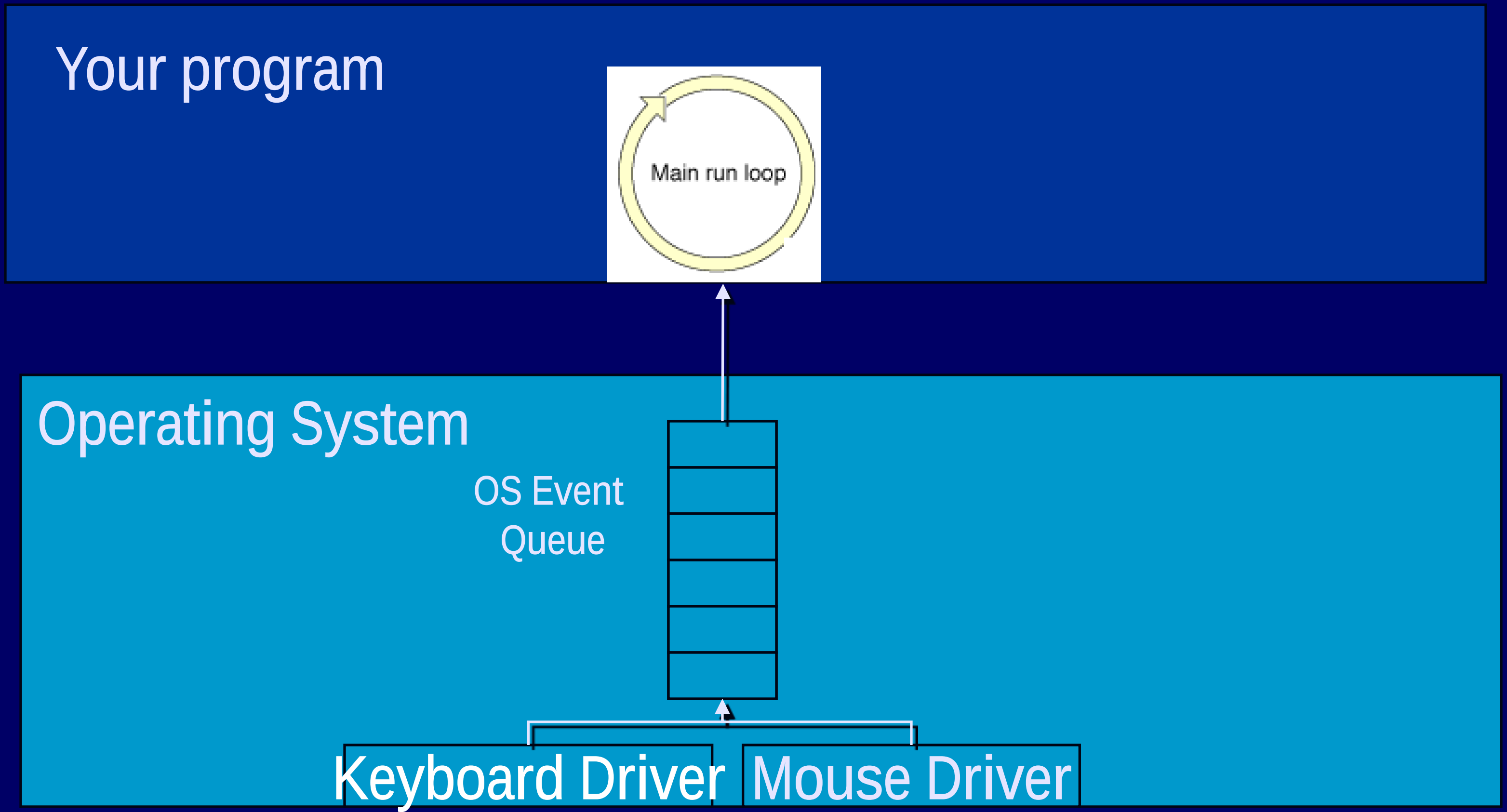
Sample: AaBbYyZz

Script: Western



Input Events: programming model

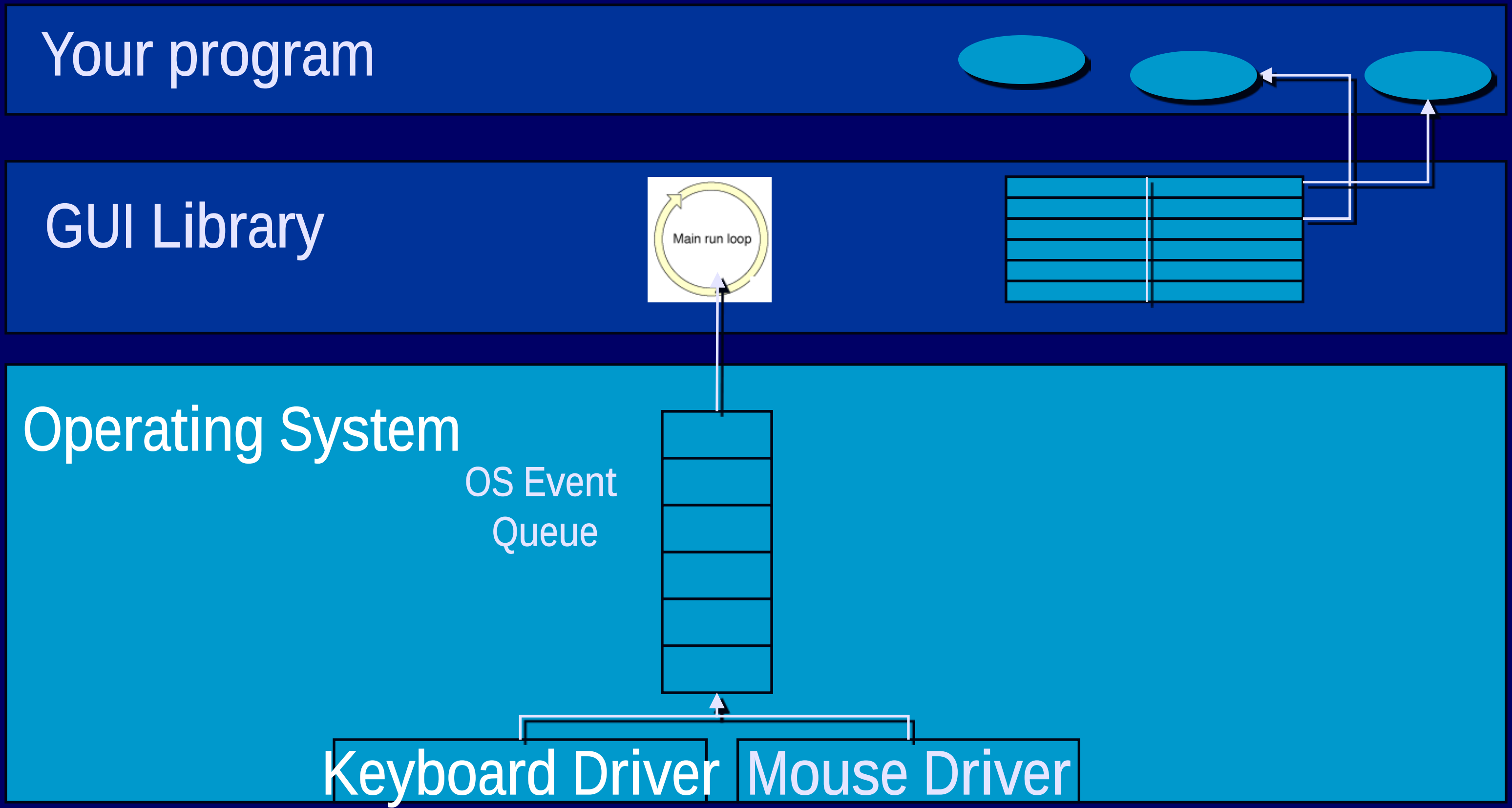
1. Use an infinite loop to keep checking the event queue
2. When you find the event you are interested in, execute the relevant code



Input Events: programming model

Use an intermediate **GUI library**:

- specify specific events you are interested in.
- specify method/function in your code that should be called when an event you are interested in is received



This Week’s To Do List

- Go through lecture slides – make sure you try the code snippets
- Try the lecture’s programs posted on course website