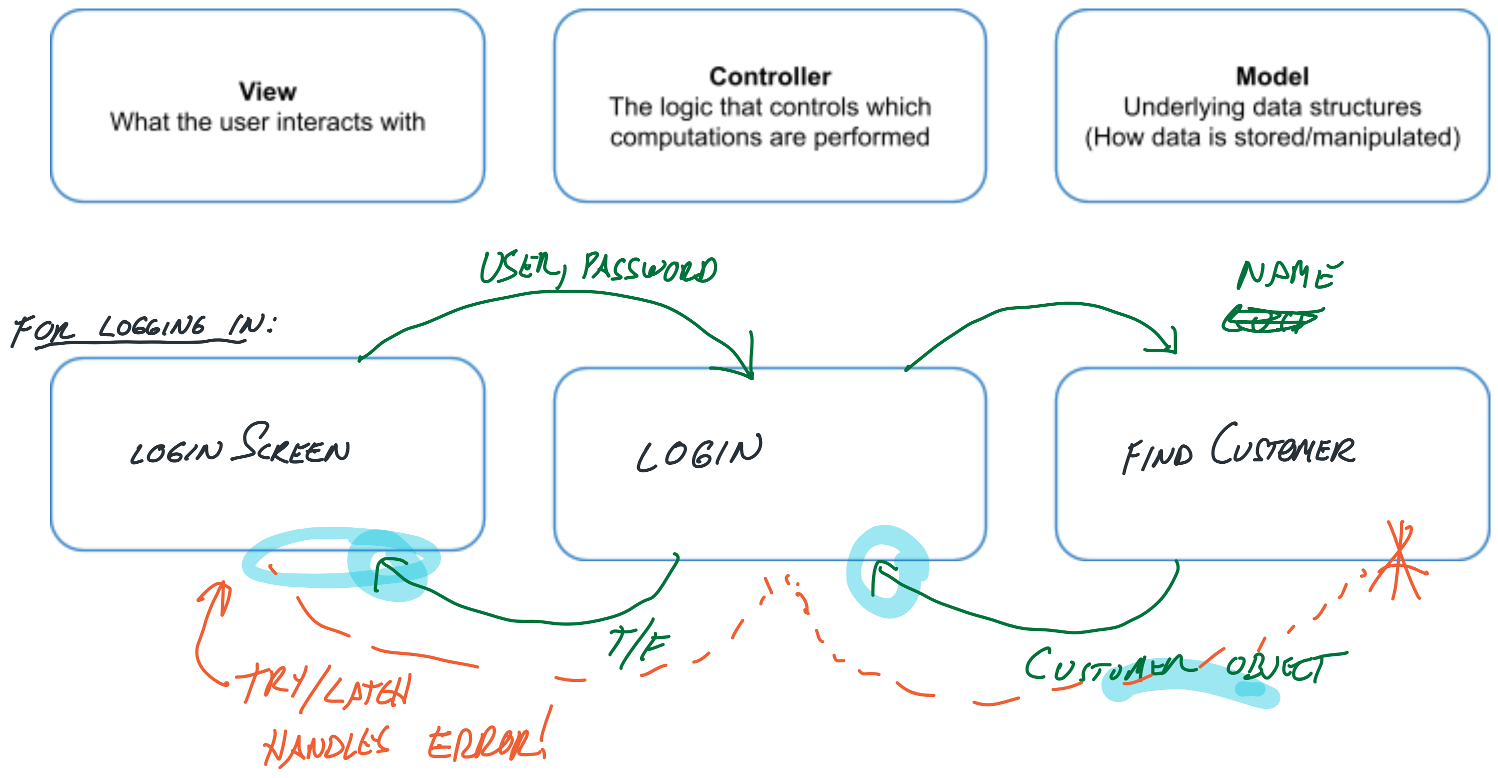
Model-View-Controller / Banking service recap


View
What the user interacts with
Controller
The logic that controls which computations are performed
Model
Underlying data structures (How data is stored/manipulated)
FOR LOGGING IN:
USER, PASSWORD
NAME
LOGIN SCREEN
LOGIN
FIND CUSTOMER
T/F
CUSTOMER OBJECT
TRY/CATCH
HANDLES ERROR!

```
// BankingConsole (view)
public void loginScreen() {
  while(loggedIn) {
      // Read user and password
      try {
          loggedIn = c.login(user, pass);
          System.out.println("Logged in");
      } catch (CustNotFoundExn e) {
          System.out.println("No such user");
      } catch (LoginFailedExn e) {
          System.out.println("Wrong password");
      }
  }
}
```

```
// CustomerList (model)
public Customer findCustomer(String custname)
    throws CustNotFoundExn {
    for (Customer cust : customers) {
        if (cust.nameMatches(custname)) {
            return cust;
        }
    }
    throw new CustNotFoundExn(custname);
}
```

```
// BankingService (controller)
public Boolean login(String custname, String withPwd)
        throws CustNotFoundExn, LoginFailedExn {
    Customer cust = customers.findCustomer(custname);
    if (cust.checkPwd(withPwd)) {
        return true;
    }
    // . . .
}
```

Key takeaways:
- MVC: can separate out user interface/data structures from program logic => flexible to change
- Exceptions: cleaner/more flexible way to handle errors separate from using “return”

Call stack: methods that are currently "outstanding" or being executed
- How programming language keeps track of where to go when a method returns

Thinking about how exceptions work

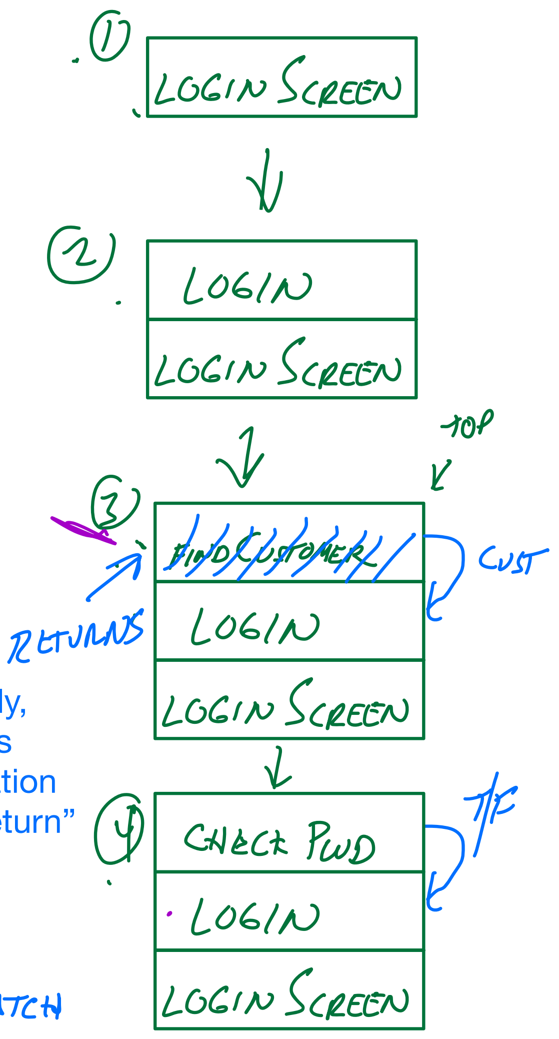
```java
// BankingConsole (view)
public void loginScreen() {
  while(loggedIn) {
      // Read user and password
      try {
          loggedIn = c.login(user, pass);
          System.out.println("Logged in");
      } catch (CustNotFoundExn e) {
          System.out.println("No such user");
      } catch (LoginFailedExn e) {
          System.out.println("Wrong password");
      }
   }
}
```

LOGIN

```java
// BankingService (controller)
public Boolean login(String custname, String withPwd)
        throws CustNotFoundExn, LoginFailedExn {
   Customer cust = customers.findCustomer(custname);
   if (cust.checkPwd(withPwd)) {
        return true;
   }
   // . . .
}
```

FIND CUSTOMER

```java
// CustomerList (model)
public Customer findCustomer(String custname)
   throws CustNotFoundExn {
   for (Customer cust : customers) {
        if (cust.nameMatches(custname)) {
            return cust;
        }
   }
   throw new CustNotFoundExn(custname);
}
```

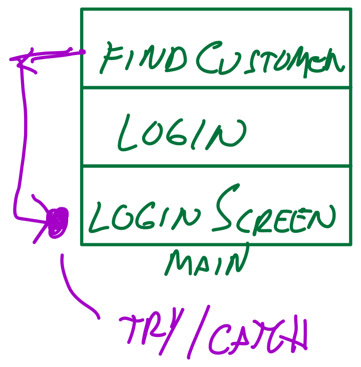


Normally, we pass information with "return"

3A What if findCustomer throws an exception?

If an exception is thrown, Java stops executing and goes back on the call stack until it reaches the nearest try/catch that can "handle" the exception

In this case,
- findCustomer throws CustNotFoundExn
- control goes immediately back to loginScreen, which is the first (only) method in the call stack that can catch CustNotFoundExn



```
public class CustomerList {
   private LinkedList<Customer> customers;

   public Customer findCustomer(String custname) throws CustNotFoundExn {
       for (Customer cust:customers) {
           if (cust.nameMatches(custname))
               return cust;
       }
       throw new CustNotFoundExn(custname); // instead of returning null
   }
}
------------------------------

public class BankingService {
   CustomerList customers;

   public Boolean login(String custname, String withPwd)
           throws CustNotFoundExn, LoginFailedExn {
       Customer cust = customers.findCustomer(custname);
       if (cust.checkPwd(withPwd)) {
           return true;
       } else {
           throw new LoginFailedExn(custname);
       }
   }
}
------------------------------

public class BankingConsole {
   BankingService controller;

   public void loginScreen() {
       boolean loggedIn = false;
       while (!loggedIn) {
           // Prompt for input
           try {
               loggedIn = controller.login(username, password);
               System.out.println("Thanks for logging in");
           } catch (CustNotFoundExn e) {
               System.out.println("No such user " + e.custname);
           } catch (LoginFailedExn e) {
               System.out.println("Password mismatch for " + e.custname);
           }
       }
   }
}
```

"while loop": loop as long as condition is true
=> Useful when you don't know how many times loop will run ahead of time (eg. how many retries the user will need)

=> In this case, keep looping so long as loggedIn is false. This lets us keep retrying logins until controller.login returns true (successful login)

(It's okay if you're not totally comfortable with while loops yet— we'll see more with them in a few weeks!)

Speeding Up Access to Customers

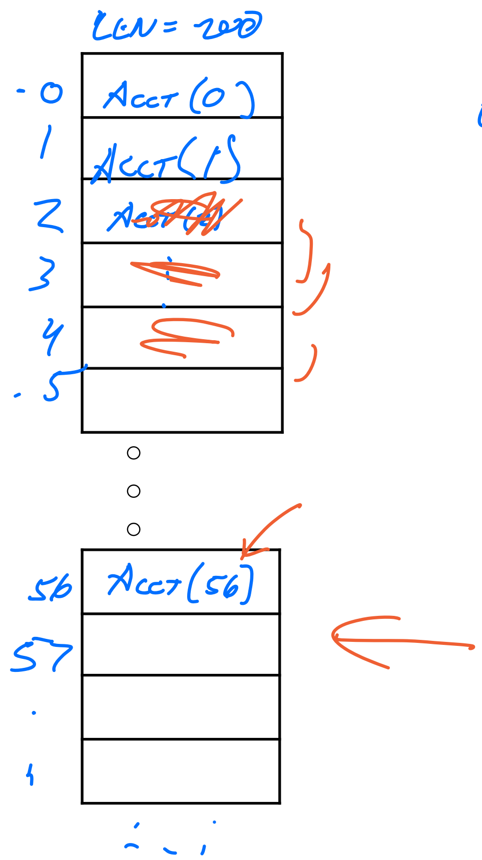
```
public class AccountList {
   private LinkedList<Account> accounts = new LinkedList<Account>();

   public Account findAccount(int forAcctNum) {
       for (Account acct:accounts) {
           if (acct.numMatches(forAcctNum))
               return acct;
       }
       return null; // not yet converted to exceptions
   }
}
```

WOULD LIKE TO BE ABLE TO LOOK UP ACCOUNTS FASTER

⇒ WANT: CONSTANT TIME. LOOKUP

WHAT IF WE USED AN ARRAY?



⇒ HOW WOULD THIS BREAK DOWN??

— EXPANDING ARRAY

— REMOVE ACCOUNTS

Want: data structure that associates account numbers with accounts that's more flexible, and can do lookups in constant time

=> HashMap (also called Dictionary, Hash table)
 - Constant time lookup of a key matched to a value

HASHMAP< INTEGER, ACCOUNT>

KEY TYPE

"VALUE"

Example for how to work with Java’s HashMap

```
public void hashMapExample() {
   HashMap<String, String> labRooms = new HashMap<>();
   labRooms.put("Mon 4-6", "CIT219");
   labRooms.put("Mon 6-8", "B&H999"); // Associate this key with this value
   labRooms.put("Tue 4-6", "CIT219"); // Multiple keys with same value OK!
   labRooms.put("Mon 4-6", "CIT317");
   // Can't have a key point to two values simultaneously!

   // . . .
   labRooms.get("Mon 6-8"); // Returns CIT219 (Where is Mon 6-8 lab?)
   labRooms.get("Mon 4-6"); // Returns CIT317 => Only one value with a
given key

   // Could also make...
   HashMap<String, List<String>> multiLabRooms = new HashMap<>();
   // multiLabRooms.put("Mon 4-6", ["X" ,"Y"]) // shorthand for list

}
```