User permission validation implemented here

```
class: state

virtual static permittedUserTypes (list of userTypes)
static enum exitCode
user activeUser
dbAccess dbHandle

bool verifyPermission(user.userType)
virtual exitCode execute(user, db handle)
```

Green highlight = things we probably need tests for

```
class: mainMenu : state

- create dbAccess file (cache db contents?)
- ask for user name and user.isValidUsername()
- query db (cache?)
- if found, assign result to activeUser, proceed, else error msg, loop
- create log obj and write it
```

Program loop:

```
- ask for  user input for next state (if invalid: error)
- create state obj
- state.execute (pass activeUser, db handle)
- test state.execute result, loop or error msg or logout

- on “logout”:
- create log obj and write it
- clear activeUser
- display daily transaction file (log)
- delete daily transaction file (log)?
- return to top
```

```
class: createUser : state

- if user not valid mode, return exitCode.accessDenied
- ask for new username and user.isValidUsername()
- ask for user type and user.isValidUserType()
- search user file (cache?), continue if not dupe
- create user obj and write to users file (and cache?)
- create log obj and write it
- return exitCode.success to main
```

```
class: deleteUser : state

- if user not valid mode, return exitCode.accessDenied
- ask for username and try user.isValidUsername()
- search user file with user obj (cache?)
- if found, delete from users file (and cache?), else error
- delete listings associated with user?
- create log obj and write it
- return exitCode.success to main
```

```
class postListing : state

- if user not valid mode, return exitCode.accessDenied
- ask for city and listing.isValidCity()
- ask for rental price and listing.isValidRentalPrice()
- ask for room # and listing.isValidNumberOfRooms()
- check if listing is locked?
- create listing obj and write to listings file (and cache?)
- to do: lock record until next session (locked list?)
- create log obj and write it
- return exitCode.success to main
```

```
class: searchListing : state

- ask for city and listing.isValidCity()
- ask for rental price
- check for * input,  else try listing.isValidRentalPrice()
- ask for room #
- check for * input, else try try listing.isValidNumberOfRooms()
- search db, present output to user
- create log obj and write it
- return exitCode.success to main
```

```
class rentListing : state

- if user not valid mode, return exitCode.accessDenied
- ask for rentalUnitID and listing.isValidRentalUnitID()
- search db for listing and store in resultingListing
- if resultListing.getRentedFlag() != 0 then (else error)
- ask for number of nights and listing.isValidNightsRented()
- present rent-per-night, total cost
- ask for confirmation, if no then return exitCode.exited
- set resultingListing.rentedFlag = true
- delete resultListing from db
- write resultListing to listings file
- create log obj and write it
- return exitCode.success to main
```

```
class: listing

listing(string) (ctor: create listing obj from string returned from db)
listing(renterID) (ctor: specify whether this obj needs a unique id)

const rentalUnitID -string
const renterID -string (default = string.empty)
city_ - string
rentalPrice_ -double
numberOfRooms_ -unsigned int
rentedFlag -bool (default = false)
nightsRented - unsigned int (default = 0)

to_string() (see spec for format)
void setCity(string) (throw IllegalArgumentException)
void setRentalPrice(string) (throw IllegalArgumentException)
void setNumberOfRooms(string) (throw IllegalArgumentException)
void setNightsRented(unsigned int) (throw Illegal ArgumentException)

string getRentalUnitID()
string getRenterID()
String getCity()
double getRentalPrice()
unsigned int getNumberOfRooms()
bool getRentedFlag()
unsigned int getNightsRented()

static bool isValidRentalUnitID(string)
static bool isValidCity(string)
static bool isValidRentalPrice(string)
static bool isValidNumberOfRooms(string)
static bool isValidNightsRented(string)
```

```
class: user

user(string) (ctor: create user obj from string returned from db)

const username_ -string
const userType_ -userType
static enum userTypes

string to_string() (see spec for format)
void setUsername(string) (throw IllegalArgumentException)
void setUserType(userType) (throw IllegalArgumentException)

string getUsername()
userType getUserType()

static bool isValidUsername(string)
static bool isValidUserType(string)
```

```
class: log

static enum transactionCodes

log(transactionCode, User, Listing- optional) (ctor)

const code (transactionCode)
const username (string)
const userType (userTypes)
const rentalID (string)
const city (string)
const numberOfBedrooms (unsigned int)
const rentalPricePerNight (double)
const numberOfNights (unsigned int)

to_string() (see spec for format)
(f fields are sent because not relevant to transaction, fill with blanks)
```

```
class: dbAccess

list<users> searchUsers(dict) (“username” = “kevin”)
bool createUser(user)
bool deleteUser(dict)
list<listings> searchListings(dict) (“city” = “Toronto”)
bool createListing(listing)
bool writeLog(log)
```

DB:
listings.txt
users.txt
log.txt

User and listing business constraint logic implemented here