Program Development

- In your CS102 project, and in many others, the basic activities are:
    - requirements – WHAT to do
    - design (UI and detailed design) – HOW to do
    - implementation – DO it
    - testing – CHECK errors and DEBUG
- They overlap and interact
- Requirements and Design stages are extremely important

Requirements

- Tasks that a program must accomplish
- What to do, *not how* to do it

Possibly

- Problem description
- Functionalities and feature lists
- Use-case descriptions

Design

- <u>How</u> a program will accomplish its requirements
  - Break the solution into manageable pieces
  - What each piece will do
  - Which classes and objects are needed, and how they will interact
  - Detailed design include how individual methods will accomplish their tasks
- UI Design
  - How will it look like? How the system will interact with the user?
  - Storyboard, illustration, description of the interactions

Implementation, Test, Debug, Maintenance

- *Implementation*: translating a design into source code
- *Testing* attempts to find errors
  - Ensure to solve the intended problem under all the constraints specified in the requirements
- *Debugging:* determining the cause of a problem and fixing it
- *Maintenance*

Coding Java Classes

```
// header

public class Person {

   // properties

   // constructors

   // methods

}
```

```
String      name;
int         age;
double      salary;
String      comments;
```

```
public Person( String   theName,
               int      theAge ) {
   name = theName;
   age = theAge;
   comments = “”;
}
```

```
public void sayName() {
   System.out.println( name);
}
```

Coding Java Classes

```
public String getName() {
    return name;
}
```

```
public String getComments() {
    return comments;
}
```

```
public void setComments( String someText) {
    comments = someText;
}
```

“get” & “set” methods for some properties (no setName!)

```
public void increaseAge() {
    age = age + 1;
}
```

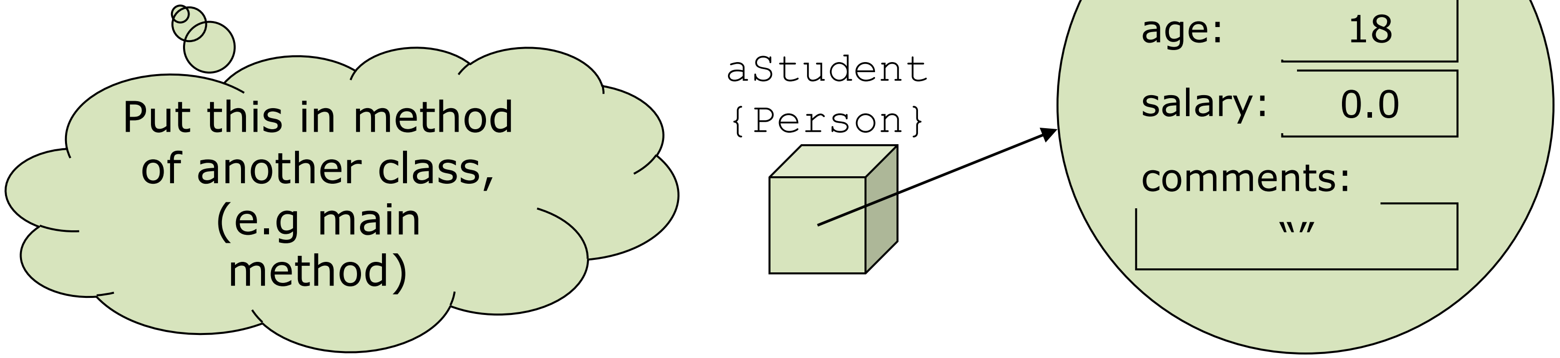
```
public double getNetSalary() {
    double netSalary;
    netSalary = salary - TAX;
    return netSalary;
}
```

Variables which are not parameters or properties must be defined locally.

Creating & Using Objects

- Always
  - Declare variable to “hold” object
  - Create object using “new” statement
  - Call object’s methods

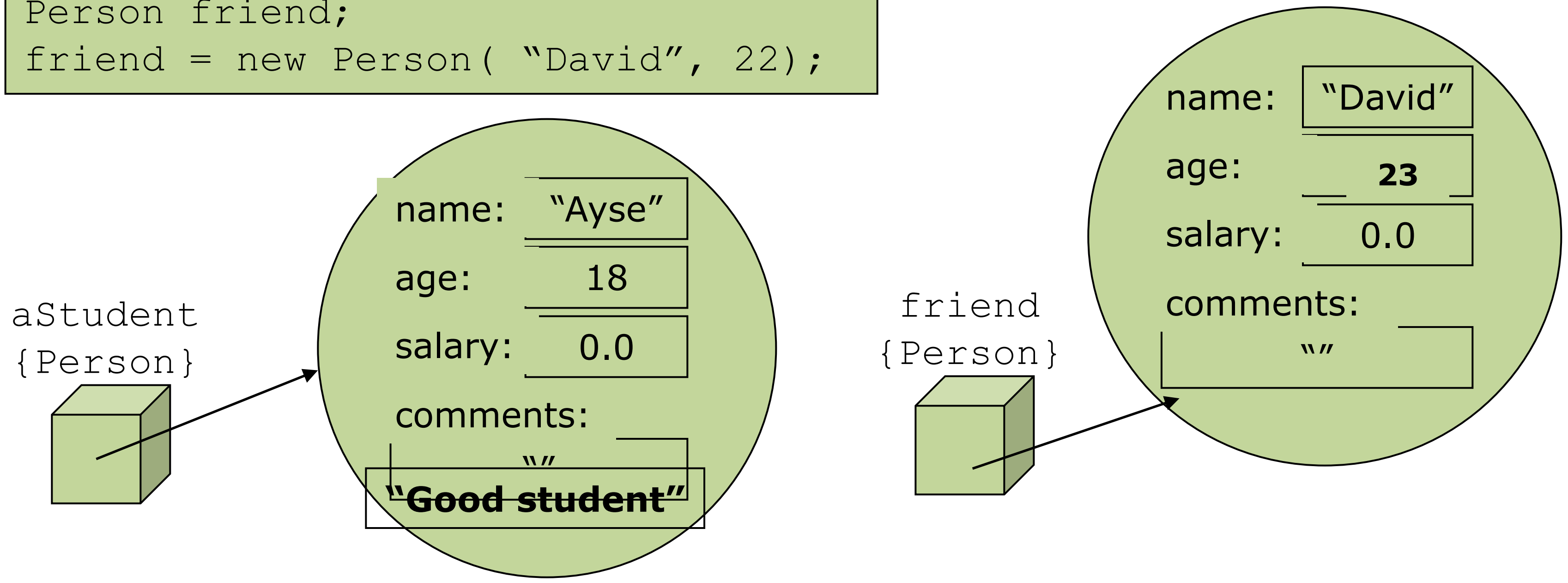
```
Person aStudent;

aStudent = new Person( “Ayse”, 18);
aStudent.sayName();
```



Creating & Using Objects

```
Person aStudent;
aStudent = new Person( "Ayse", 18);
```

```
Person friend;
friend = new Person( "David", 22);
```



```
friend.increaseAge();
aStudent.setComments( "Good student");
```

Identifying Classes and Objects

- The core activity: Determine the classes and objects
- Reuse the classes (a class library, etc.)
- One way to identify potential classes is to identify the objects discussed in the requirements
- Objects are generally nouns, and the services that an object provides are generally verbs

Identifying Classes and Objects

- A partial requirements document:

The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

Of course, not all nouns will correspond to a class or object in the final solution

Identifying Classes and Objects

- Remember: A class is a concept for a group (classification) of objects with the same behaviors
  - Singular nouns: `Coin`, `Student`, `Employee`
- Need to decide whether something should be represented as a class
  - Should Address of an `employee` be an instance variable or an object itself?
- When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

Identifying Classes and Objects

- Define the classes with the proper amount of detail
- May be unnecessary to create separate classes for each type of appliance in a house
  - Sufficient to define a more general `Appliance` class with appropriate instance data
- It all depends on the details of the problem being solved

Example: A Card Game*

- Design & implement a program to play a simple game of cards between four players. To begin, a full pack of cards are shuffled and dealt face-down to the players. The game then proceeds in rounds. For each round, players play the top card from their hand and add it face-up to a pile on the table in front of them. The player who plays the highest value card is the winner of the round and their score is incremented by one. When all of the cards have been played the player with the highest score is declared the winner of the game.

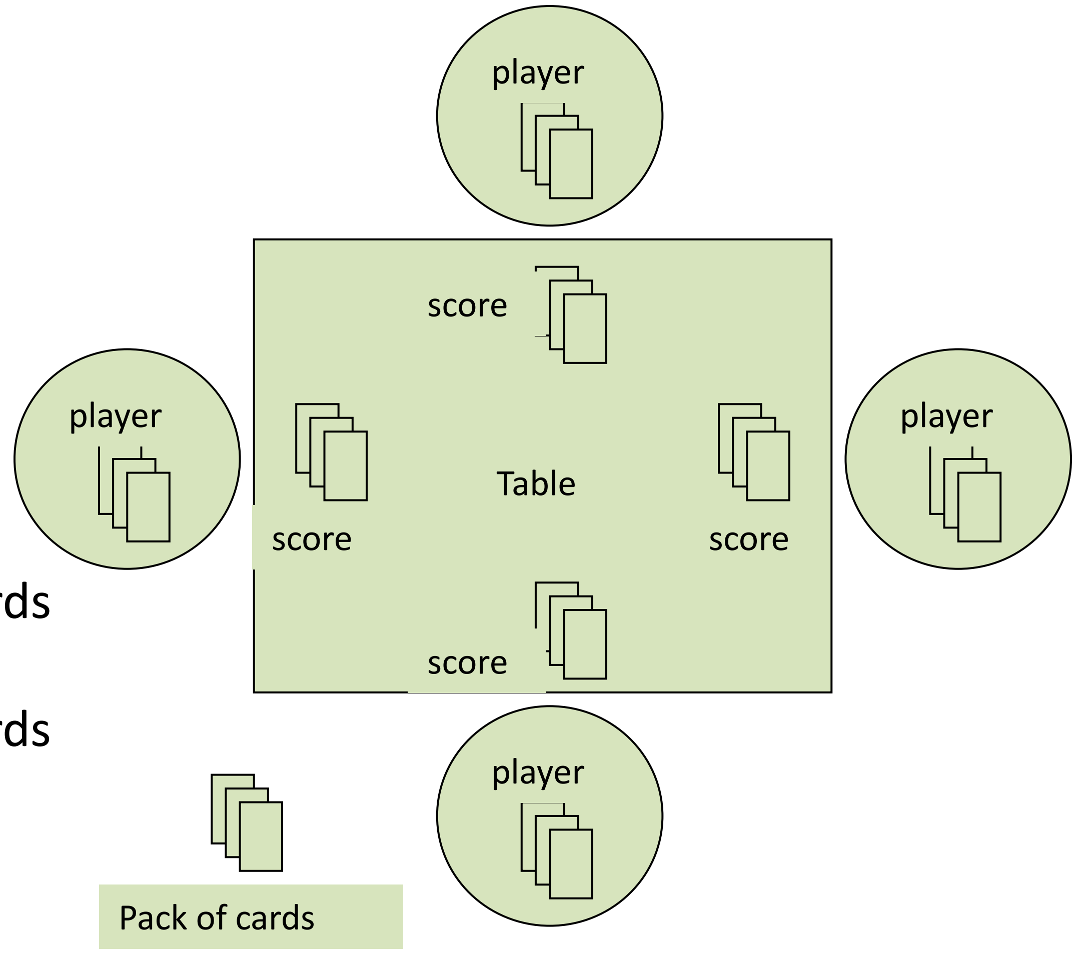
*** by David Davenport**

Example: A Card Game

- Design & implement a program to play a simple *game of cards* between four *players*. To begin, a *full pack of cards* are shuffled and dealt face-down to the players. The game then proceeds in rounds. For each round, *players* play the top *card* from their *hand* and add it face-up to a *pile* on the *table* in front of them. The *player* who plays the highest value *card* is the winner of the round and their *score* is incremented by one. When all of the *cards* have been played the *player* with the highest score is declared the winner of the game.

Picture it…

- **Objects**
  - the Game?
  - Pack of cards
  - Players 1, 2, 3 & 4
  - the Table?
  - Score card? (Player 1, 2, 3 & 4 scores)
  - Players 1, 2, 3 & 4 Cards on table
  - Players 1, 2, 3 & 4 Cards in hand



Classes

- **CleverGame**
  - Pack of cards
  - 4 Players
  - Scorecard
    (with 4 scores on)
  - 4 Piles of cards
    on table
- **ScoreCard**
  - Set of scores
- **Player**
  - *Name*
  - Set of cards in hand
- **Cards**
  - Collection of cards
- **Card**
  - Face value
  - *Suit*

Example: A Card Game*

- Design & implement a program to play a simple *game of cards* between four *players*. To begin, a *full pack of cards* are shuffled and dealt face-down to the players. The game then proceeds in rounds. For each round, *players* play the top *card* from their *hand* and add it face-up to a *pile* on the *table* in front of them. The *player* who plays the highest value *card* is the winner of the round and their score is incremented by one. When all of the *cards* have been played the *player* with the highest score is declared the winner of the game.

CleverGame class

- **<u>properties</u>**
  - Pack of cards, 4 Players
  - ScoreCard, 4 Piles of cards on table
- **<u>constructor</u>** (4 players)
  creates the game with the given players
- **<u>Methods</u>**
  - + playTurn(Player, Card) : boolean
  - + isTurnOf(Player) : boolean
  - + isGameOver() : boolean
  - + getScore(playerNumber) : int
  - + getName(playerNumber) : String
  - + getRoundNo() : int
  - + getTurnOfPlayerNo() : int
  - + getWinners() : set of Player

Represents a single card game played by 4 players

Player class

- **properties**
  - *name*
  - set of cards in hand
- **constructor** (name)
  creates player with name & empty hand
- **methods**
  - getName()
    returns players name
  - add(Card)
    add the card to players hand
  - playCard()
    removes and returns the top card from the players hand

Represents a single player for a card game

ScoreCard class

Represents a ScoreCard for a card game

- **properties**
  - Set of scores
- **constructor** (noOfScores)
  initialises scorecard with noOfScores entries all set to zero
- **methods**
  - + getScore(scoreNo) : int
    returns specified score
  - + update(scoreNo, amount)
    add amount to scoreNo

Cards class

- **properties**
  - Collection of cards
- **constructor** ()
  creates a collection of cards with no cards in it!
- **methods**
  - getTopCard()
    removes & returns top card from collection
  - addTopCard(Card)
    adds the card to the collection
  - createFullPackOfCards()
  - shuffle()
    randomises order of cards in collection

Represents a set of zero or more playing cards

Card class

- **<u>properties</u>**
  - faceValue
  - *suit*

Represents a single playing card

- **<u>constructor</u>** (faceValue, suit)
  creates card with given face value & suit
  <u>constructor</u> (cardNumber)
  creates card with given position number in ordered pack!

- **<u>methods</u>**
  - getFaceValue()
    returns faceValue
  - getSuit()
    returns suit
  - toString()

Playing the Game

- Algorithm for playGame method
  - *Create the pack of cards*
  - *Shuffle the pack*
  - *Deal all the pack between the players*
  - *Create empty piles of cards on table*
  - *Set all players scores to zero*
  - *For each round (until players have no cards left)*
    - *Each player plays card by adding it to their pile on table*
    - *Find biggest value card on top of piles on table*
    - *Increment scores of players who played cards with biggest value*