Singleton Design Pattern

Singleton

- Intent
  - Guarantee that a class has only one instance.

-
```
public class Singleton{
    private Singleton(){};
    private static Singleton instance = null;

    public static Singleton getInstance(){
          if(instance==null)
                instance = new Singleton ();
          return instance;
    }
}
```

- You should not define public constructors.
- If you do not define constructors...

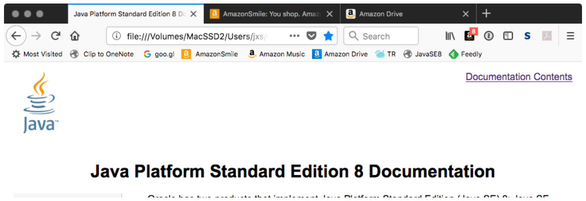
-
```
Singleton instance = Singleton.getInstance();
instance.hashCode();
Singleton instance = Singleton.getInstance();
instance.hashCode();
```

- hashCode() returns a unique ID for each instance.
  - Different instances of a class have different IDs.

- *Singleton* is an application of *Static Factory Method*.
  - getInstance() is a static factory method.
  - *Singleton* focuses on a requirement to have a class keep only one instance.

What Can be a Singleton?

- Object pools
  - Pool of objects such as browser tabs, files, threads and DB connections.
- Logger
- Plugin manager
- Access counter
- Game loop
- …, etc.



Java Platform Standard Edition 8 Documentation

Oracle has two products that implement Java Platform Standard Edition (Java SE) 8: Java SE

Alternative Implementation with Null Checking API in Java 7

- `java.util.Objects`, extending `java.lang.Object`
  - – A utility class (i.e., a set of static methods) for the instances of `java.lang.Object` and its subclasses.
  - –
    ```
    class Foo{
        private String str;
        public Foo()(String str){
            this.str = Objects.requireNonNull(str);
        } }
    ```
    - `requireNonNull()` throws a `NullPointerException` if `str==null`. Otherwise, it simply returns `str`.
  - –
    ```
    class Foo{
        private String str;
        public Foo()(String str){
            this.str = Objects.requireNonNull(
                              str, "str must be non-null!!!");
        } }
    ```
    - `requireNonNull()` can accept an error message, which is to be contained in a `NullPointerException`.

- Traditional null checking
  - –
    ```
    if(str == null)
        throw new NullPointerException();
    this.str = str;
    ```
- With `Objects.requireNonNull()`
  - – `this.str = Objects.requireNonNull(str);`
- Can eliminate an explicit conditional statement and make code a bit simpler.

Singleton Implementation with `Objects.requireNonNull()`

-
```
public class SingletonNullCheckingJava7{
    private SingletonNullCheckingJava7(){};
    private static SingletonNullCheckingJava7 instance = null;

    public static SingletonNullCheckingJava7 getInstance(){
      try{
          return Objects.requireNonNull(instance);
      }
      catch(NullPointerException ex){
          instance = new SingletonNullCheckingJava7();
          return instance;
    } }
```

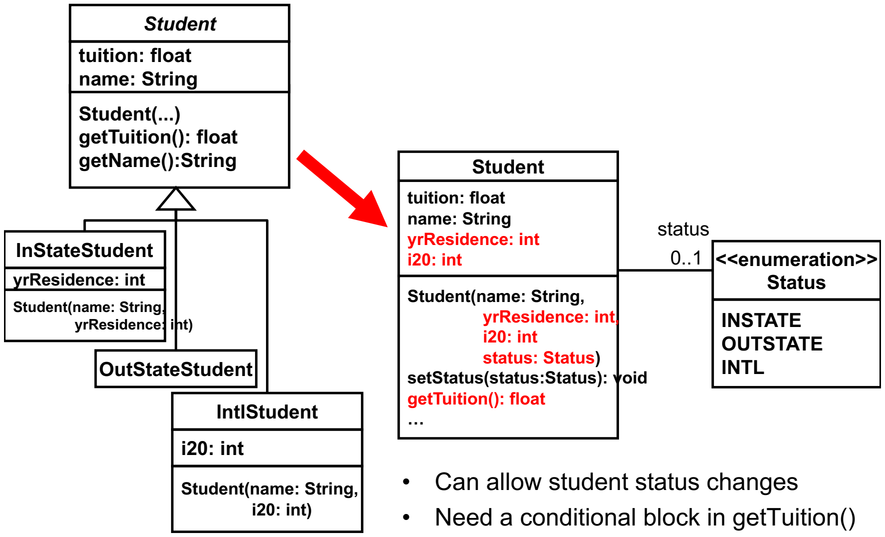
-
```
public class Singleton{
    private Singleton(){};
    private static Singleton instance = null;

    public static Singleton getInstance(){
           if(instance==null)
                  instance = new Singleton ();
           return instance;
    } }
```

State Design Pattern

State

- Intent
  - Allow an object to change its behavior according to its state.

Eliminating Class Inheritance



Student
tuition: float
name: String
Student(...)
getTuition(): float
getName():String

InStateStudent
yrResidence: int
Student(name: String, yrResidence: int)

OutStateStudent

IntlStudent
i20: int
Student(name: String, i20: int)

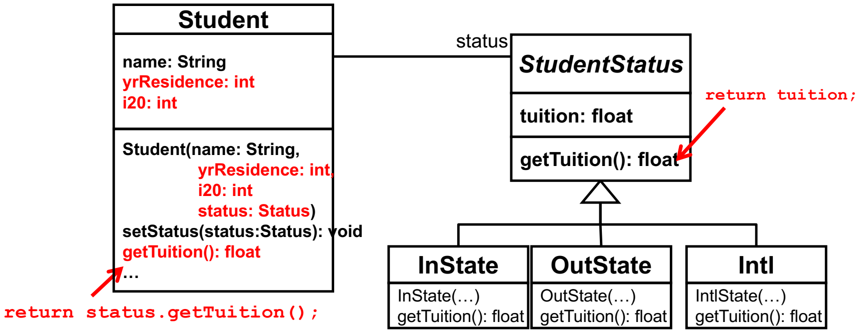
Student
tuition: float
name: String
yrResidence: int
i20: int
Student(name: String, yrResidence: int, i20: int status: Status)
setStatus(status:Status): void
getTuition(): float
...

status
0..1

<<enumeration>>
Status
INSTATE
OUTSTATE
INTL

- Can allow student status changes
- Need a conditional block in getTuition()
  - Can remove the conditional with *State*.

Design Improvement with *State*



Student
name: String
yrResidence: int
i20: int
Student(name: String, yrResidence: int, i20: int status: Status)
setStatus(status:Status): void
getTuition(): float
...

`return status.getTuition();`

status

StudentStatus
tuition: float
getTuition(): float

`return tuition;`

InState
InState(…)
getTuition(): float

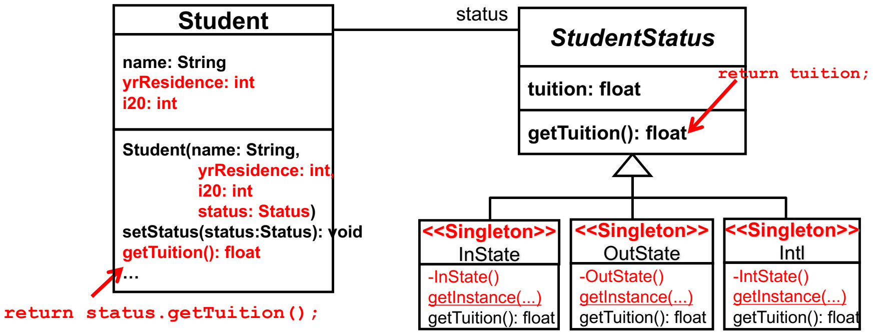
OutState
OutState(…)
getTuition(): float

Intl
IntlState(…)
getTuition(): float

```
Student s1 = new Student( ..., new InState(...) );
s1.getTuition();
```

c.f. lecture note #3

State Classes as *Singleton*



Student
name: String
yrResidence: int
i20: int
Student(name: String, yrResidence: int, i20: int status: Status)
setStatus(status:Status): void
getTuition(): float
...

`return status.getTuition();`

status

StudentStatus
tuition: float
getTuition(): float

`return tuition;`

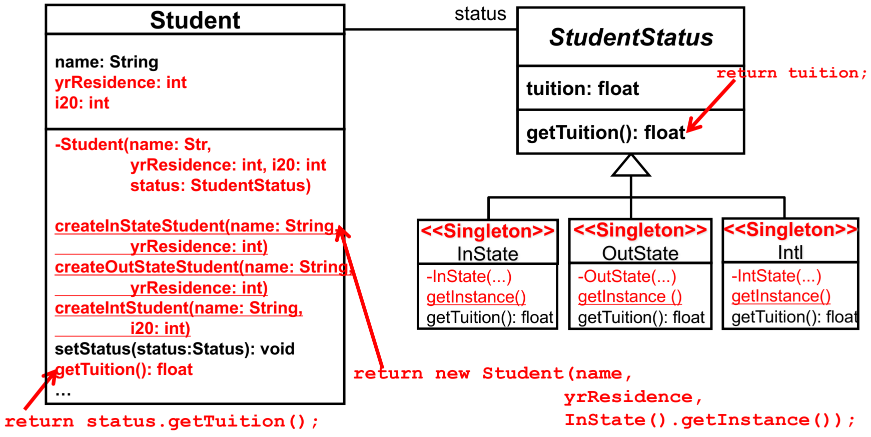
<<Singleton>>
InState
-InState()
getInstance(...)
getTuition(): float

<<Singleton>>
OutState
-OutState()
getInstance(...)
getTuition(): float

<<Singleton>>
Intl
-IntState()
getInstance(...)
getTuition(): float

```
Student s1 = new Student( ..., InState.getInstance(...) );
s1.getTuition();
```

Adding *Static Factory Methods*



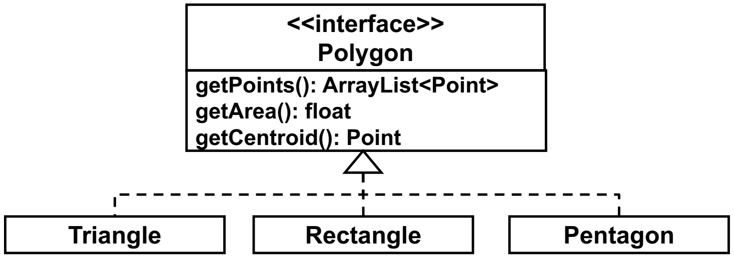
```
return tuition;
```

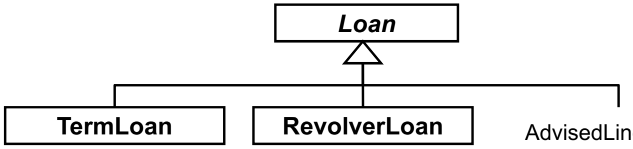
```
return status.getTuition();
```

```
return new Student(name,
                   yrResidence,
                   InState().getInstance());
```

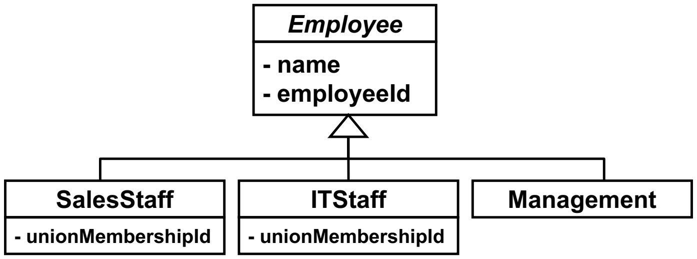
```
Student s1 = Student.createInStateStudent( "John Smith", 18 );
s1.getTuition();
```

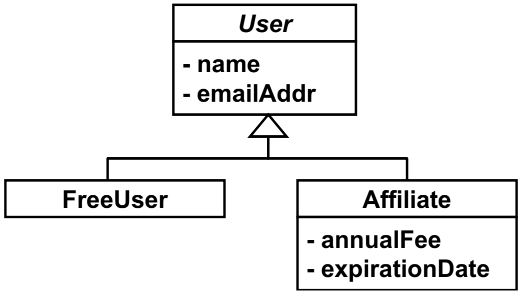
Other Use Cases





AdvisedLineLoan
SPLC
RCTL
risk adjusted versions
etc.





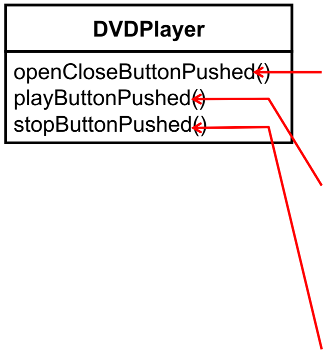
Another Example: DVD Player

- Imagine you implement a firmware of DVD players
  - Focus on a player’s ***behaviors*** upon ***events***.
  - Events
    - The “Open/Close” button is pushed.
    - The “Play” button is pushed.
    - The “Stop” button is pushed.
  - The player differently behaves upon a certain event depending on its current state.
    - State-dependent behaviors

State-Dependent Behaviors

- When the “open/close” button pushed,
  - Opens the drawer
    - If the drawer is closed and the player is not playing a DVD.
  - Stops playing a DVD and opens the drawer
    - if the drawer is closed and the player is playing a DVD.
  - Closes the drawer
    - if the drawer is open.
- When the “play” button pushed,
  - Plays a DVD
    - If the drawer is closed.
      - Displays an error message if the drawer is empty.
  - Closes the drawer and plays a DVD
    - If the drawer is open.
      - Displays an error message if the drawer is empty.
- When the “stop” button pushed
  - Stops playing a DVD
    - If the drawer is closed and the player is playing a DVD
  - Does nothing.
    - If the drawer is closed and the player is not playing a DVD.
  - Does nothing
    - If the drawer is open.

How to Implement State-dependent Behaviors?



DVDPlayer

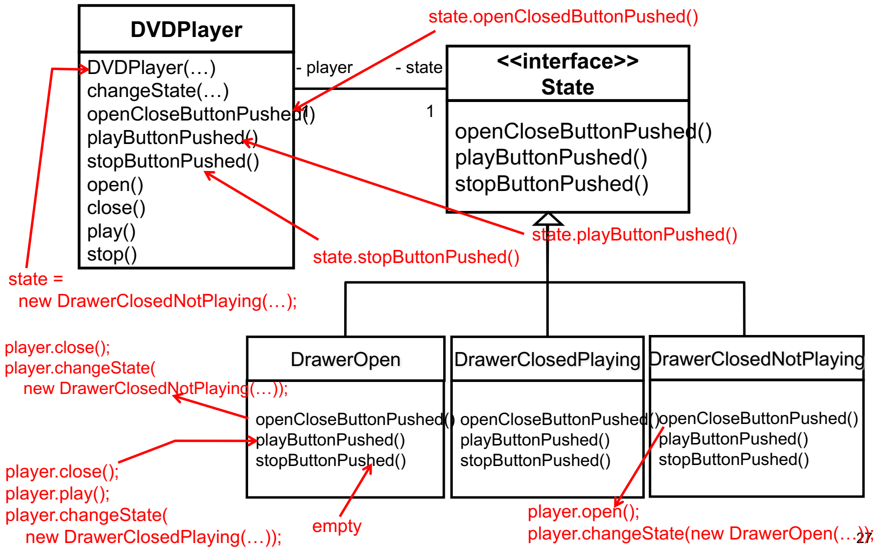
openCloseButtonPushed()
playButtonPushed()
stopButtonPushed()

If the drawer is closed and the player is not playing a DVD
Open the drawer
If the drawer is closed and the player is playing a DVD
Stops playing a DVD and open the drawer
If the drawer is open
Close the drawer

If the drawer is closed
Play a DVD
If the drawer is open
Close the drawer and play a DVD.

If …
Stop playing a DVD
If …
Do nothing.
If …
Do nothing

Defining States as Classes



DVDPlayer

DVDPlayer(…)
changeState(…)
openCloseButtonPushed()
playButtonPushed()
stopButtonPushed()
open()
close()
play()
stop()

- player

- state

1

<<interface>>
State

openCloseButtonPushed()
playButtonPushed()
stopButtonPushed()

DrawerOpen

openCloseButtonPushed()
playButtonPushed()
stopButtonPushed()

DrawerClosedPlaying

openCloseButtonPushed()
playButtonPushed()
stopButtonPushed()

DrawerClosedNotPlaying

openCloseButtonPushed()
playButtonPushed()
stopButtonPushed()

state.openClosedButtonPushed()

state.playButtonPushed()

state.stopButtonPushed()

```
state =
  new DrawerClosedNotPlaying(…);
```

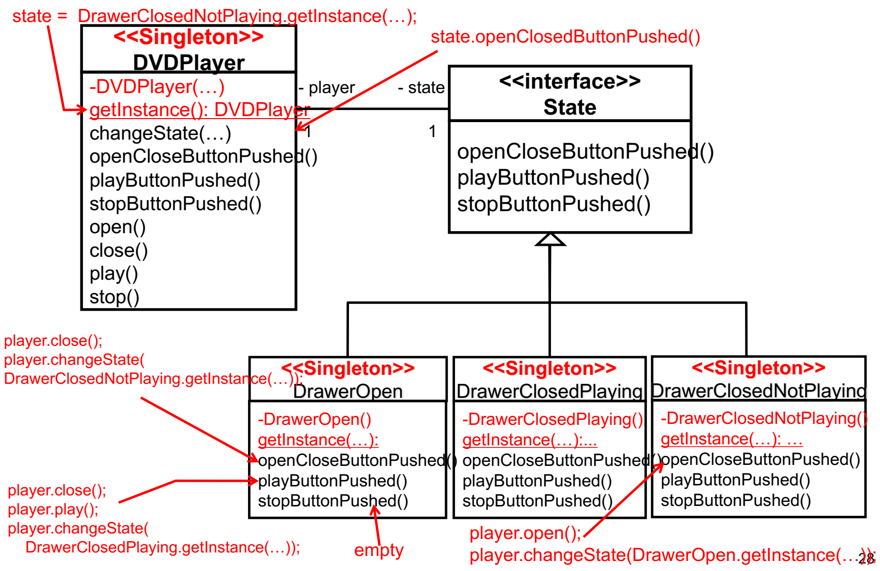
```
player.close();
player.changeState(
   new DrawerClosedNotPlaying(…));
```

```
player.close();
player.play();
player.changeState(
   new DrawerClosedPlaying(…));
```

empty

```
player.open();
player.changeState(new DrawerOpen(…));
```

State Classes as *Singleton*



state = DrawerClosedNotPlaying.getInstance(…);

state.openClosedButtonPushed()

<<Singleton>>
DVDPlayer

-DVDPlayer(…)
getInstance(): DVDPlayer
changeState(…)
openCloseButtonPushed()
playButtonPushed()
stopButtonPushed()
open()
close()
play()
stop()

- player
1

- state
1

<<interface>>
State

openCloseButtonPushed()
playButtonPushed()
stopButtonPushed()

<<Singleton>>
DrawerOpen

-DrawerOpen()
getInstance(…):
openCloseButtonPushed()
playButtonPushed()
stopButtonPushed()

<<Singleton>>
DrawerClosedPlaying

-DrawerClosedPlaying()
getInstance(…):...
openCloseButtonPushed()
playButtonPushed()
stopButtonPushed()

<<Singleton>>
DrawerClosedNotPlaying

-DrawerClosedNotPlaying()
getInstance(…): …
openCloseButtonPushed()
playButtonPushed()
stopButtonPushed()

player.close();
player.changeState(
DrawerClosedNotPlaying.getInstance(…));

player.close();
player.play();
player.changeState(
DrawerClosedPlaying.getInstance(…));

empty

player.open();
player.changeState(DrawerOpen.getInstance(…));

If-based and State-based Designs

- If-based
  - Easy to implement first
  - Hard to maintain a long sequence of conditional branches (“if” statements).
- State-based
  - May not be that easy to implement first
  - Easy to maintain
    - If new buttons/events are added, just add extra methods in state classes.
      - No need to modify existing methods.
    - If new states are added, just add extra state classes.
      - No need to modify existing classes.
  - Initial cost may be higher, but maintenance cost (or total cost) should be lower
    - as changes are made in the future.

State

- Intent
  - Allow an object to change its behavior according to its state.
  - Can implement state-dependent behaviors without conditionals.

HW 3

- Implement the design in Slide 25.
- Write and run at least one test case for every single public method.
- Deadline: March 8 (Thu) midnight
  - Start working on it early. Expect a potential “dealth by XML.”

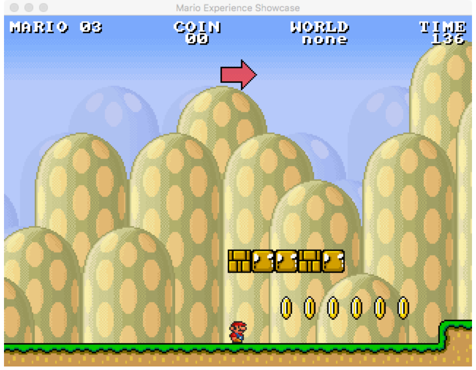
- Use <batchtest> to have Ant search test classes in your project directory and run all of them (DVDPlayerTest, etc.).

```
<junit …>
    …
    <batchtest …>
    ...
    </batchtest>
    ...
</junit>
```

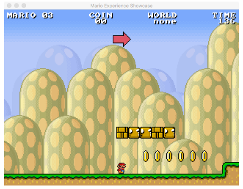
  - c.f. JUnit documentation

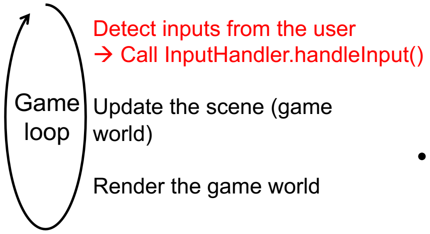
One More Example: A Simple 2D Game

- Imagine a simple 2D game like Super Mario
  - http://www.marioai.org/
  - http://www.platformersai.com



Handling User Inputs





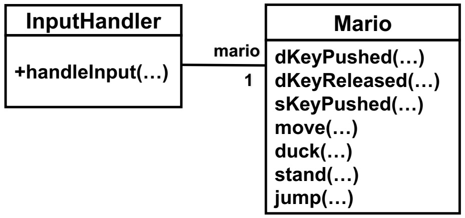
Detect inputs from the user
→ Call InputHandler.handleInput()

Update the scene (game world)

Render the game world

```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput(...);
  ...
}
```

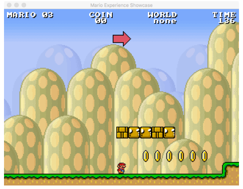
- 5 types of inputs
  - The user pushes the right arrow, left arrow, down arrow and “s” keys.
    - R arrow to move right
    - L arrow to move left
    - D arrow to duck
    - “s” to jump
  - The user releases the D arrow to stand up.
- InputHandler
  - handleInput()
    - identifies a keyboard input since the last game loop iteration (i.e. since the last frame).
    - 60 frames/s (FPS): One input per frame (i.e. during 1.6 msec)

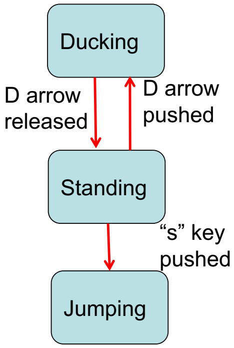


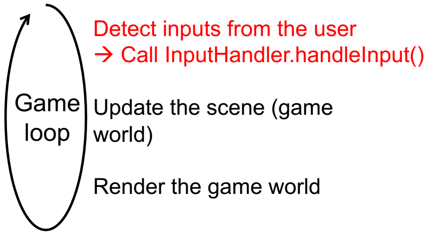
InputHandler
+handleInput(…)

mario 1

Mario
dKeyPushed(…)
dKeyReleased(…)
sKeyPushed(…)
move(…)
duck(…)
stand(…)
jump(…)





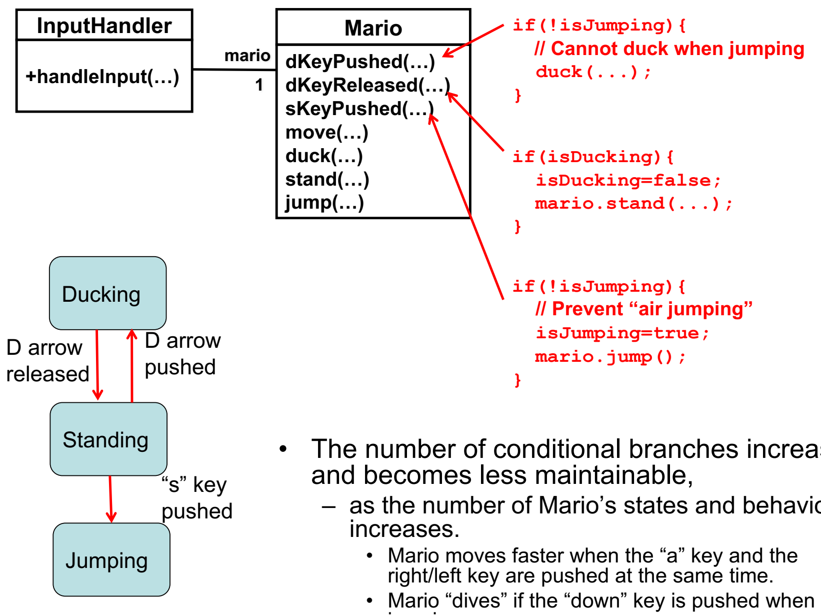


Detect inputs from the user
→ Call InputHandler.handleInput()

Update the scene (game world)

Render the game world

```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput(...);
  ...
}
```

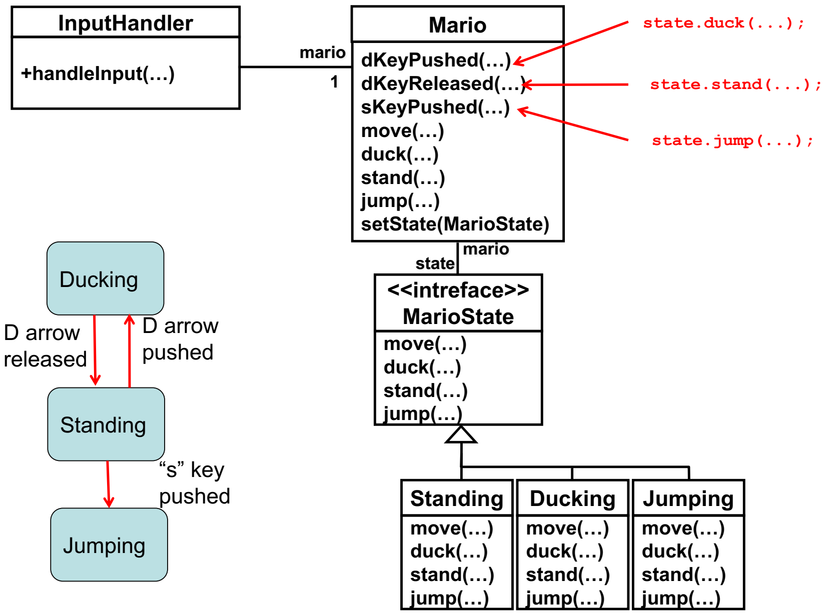


```
if(!isJumping){
   // Cannot duck when jumping
   duck(...);
}
```

```
if(isDucking){
   isDucking=false;
   mario.stand(...);
}
```

```
if(!isJumping){
   // Prevent “air jumping”
   isJumping=true;
   mario.jump();
}
```

- The number of conditional branches increases and becomes less maintainable,
  - as the number of Mario’s states and behaviors increases.
    - Mario moves faster when the “a” key and the right/left key are pushed at the same time.
    - Mario “dives” if the “down” key is pushed when jumping.



HW 4

- Explain how each state class’s methods should be implemented.

- Deadline: