Ollscoil na Gaillimhe
University of Galway

CT2106
Object Oriented Programming

Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@University*of*Galway.ie

School of Computer Science

University
*of*Galway.ie

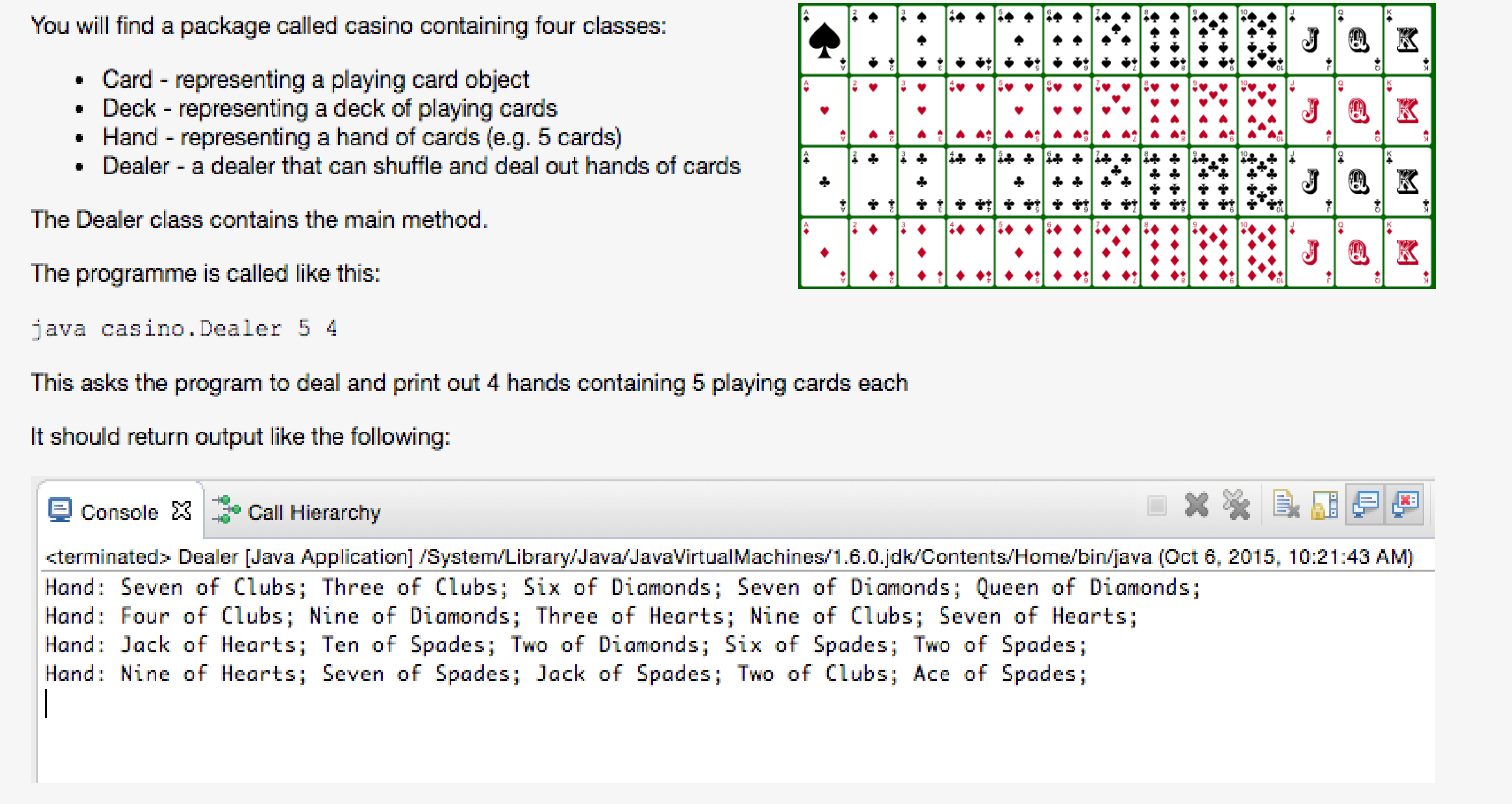
Todays Topics

- Static Fields
- Exceptions

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Some of the examples in this lecture are from Allen Downey’s book: “**Think Java v6**”

Card assignment from a previous year



You will find a package called casino containing four classes:

- Card - representing a playing card object
- Deck - representing a deck of playing cards
- Hand - representing a hand of cards (e.g. 5 cards)
- Dealer - a dealer that can shuffle and deal out hands of cards

The Dealer class contains the main method.

The programme is called like this:

```
java casino.Dealer 5 4
```

This asks the program to deal and print out 4 hands containing 5 playing cards each

It should return output like the following:

```
<terminated> Dealer [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Oct 6, 2015, 10:21:43 AM)
Hand: Seven of Clubs; Three of Clubs; Six of Diamonds; Seven of Diamonds; Queen of Diamonds;
Hand: Four of Clubs; Nine of Diamonds; Three of Hearts; Nine of Clubs; Seven of Hearts;
Hand: Jack of Hearts; Ten of Spades; Two of Diamonds; Six of Spades; Two of Spades;
Hand: Nine of Hearts; Seven of Spades; Jack of Spades; Two of Clubs; Ace of Spades;
```

Details

- A card game involves cards of different values
- These are normally gathered together in a Deck
- There are a number of things you might want to do with a deck

  Shuffle the deck

  Deal the deck

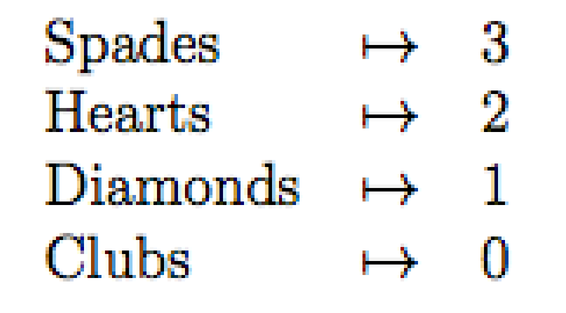
  Sort the deck

  Search for a card

The Card Class

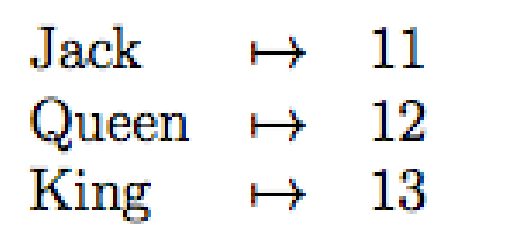
Each Card has a **suit** and a **rank** –represented as instance variables.

suit



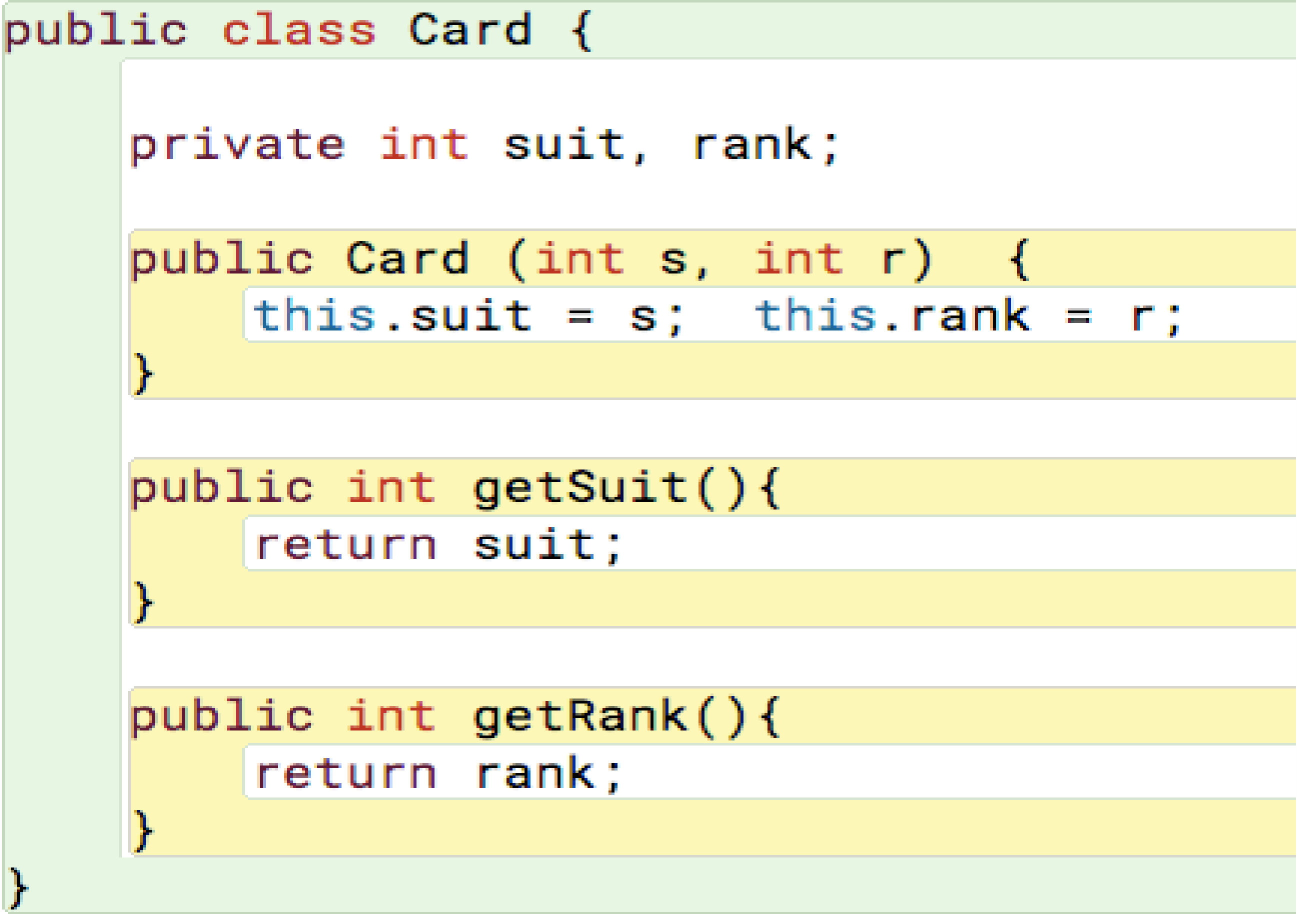
Spades	↦	3
Hearts	↦	2
Diamonds	↦	1
Clubs	↦	0

rank



Jack	↦	11
Queen	↦	12
King	↦	13

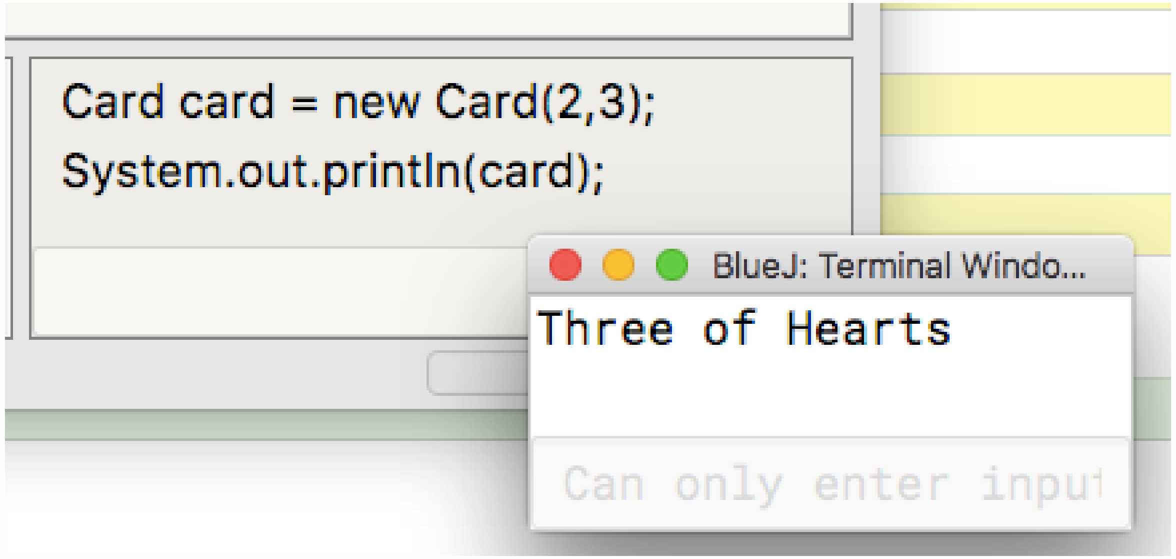
Simple Card Class



```
public class Card {

    private int suit, rank;

    public Card (int s, int r)  {
        this.suit = s;  this.rank = r;
    }

    public int getSuit(){
        return suit;
    }

    public int getRank(){
        return rank;
    }
}
```

Card Class

What if you want to be able to print out the value of this Card using the **toString()** method E.g



```
Card card = new Card(2,3);
System.out.println(card);
```

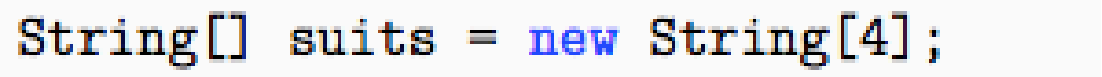
```
Three of Hearts
```

Linking suit and rank

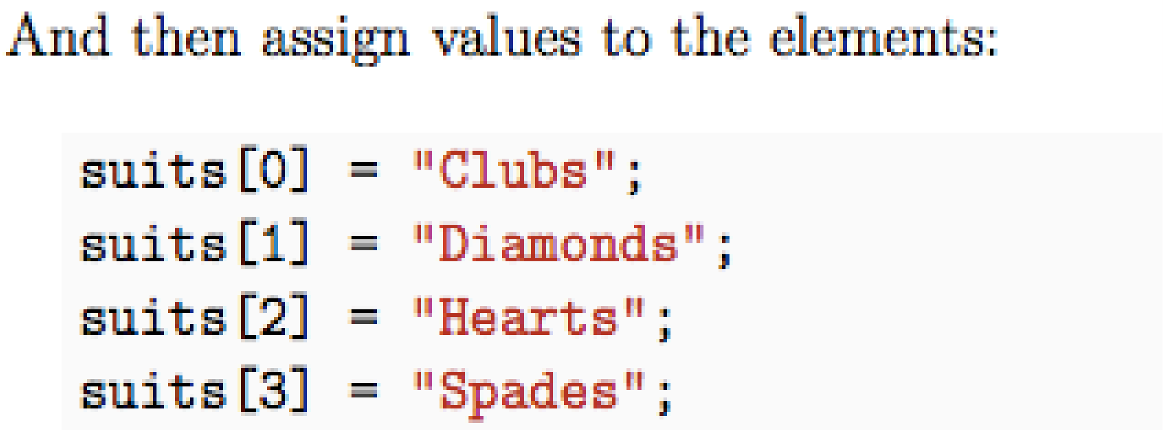
We need to link the suit and rank *int* values to the String values representing the Card

Suit: 2 - > “Hearts”
Rank: 3 -> “Three”

You can declare an array of Strings to hold all possible rank values



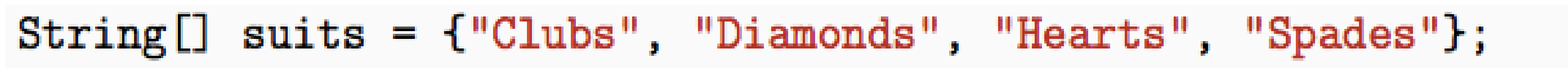
```
String[] suits = new String[4];
```



And then assign values to the elements:

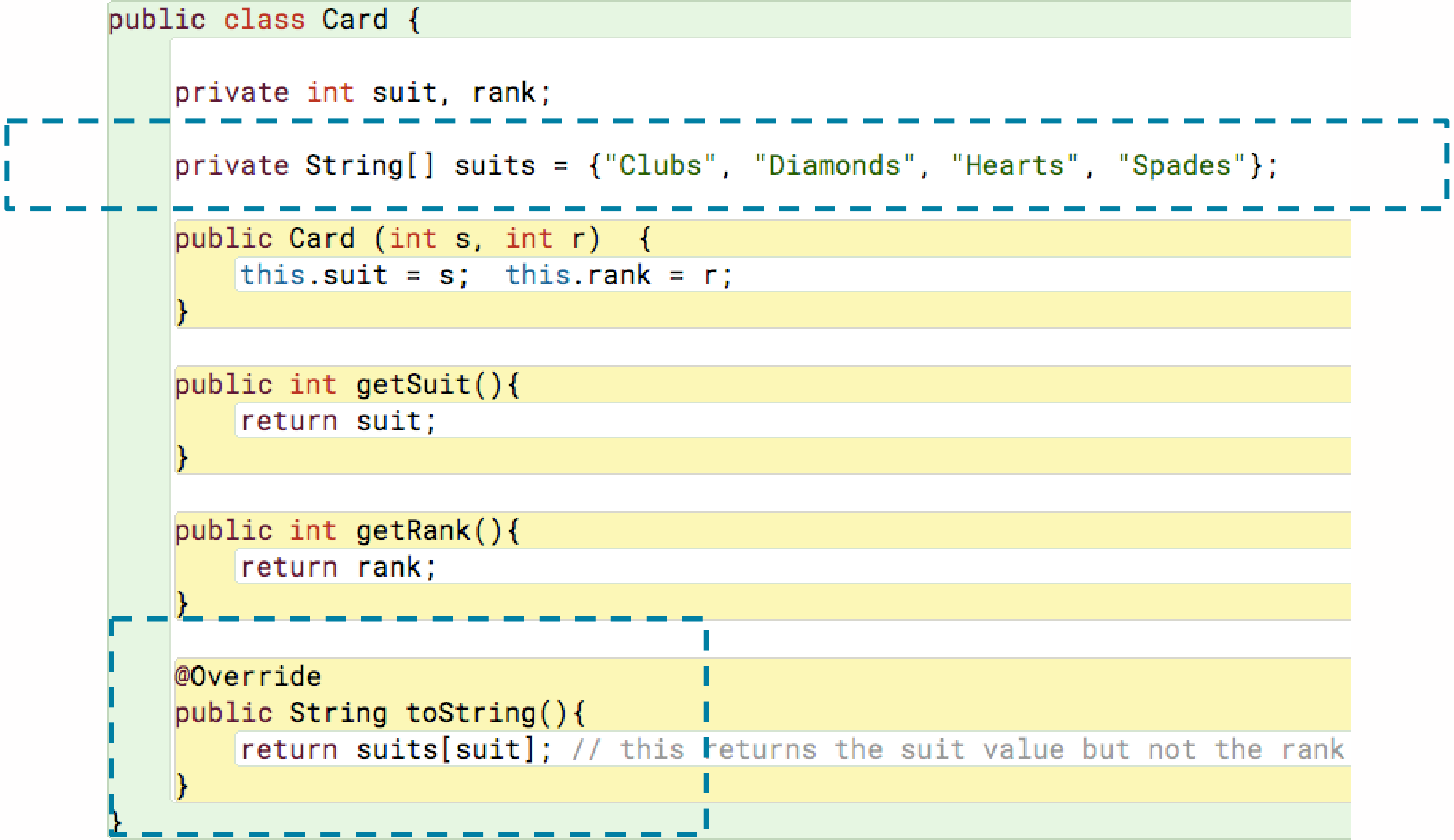
```
suits[0] = "Clubs";
suits[1] = "Diamonds";
suits[2] = "Hearts";
suits[3] = "Spades";
```

Or you can declare and assign values all in one go



```
String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
```

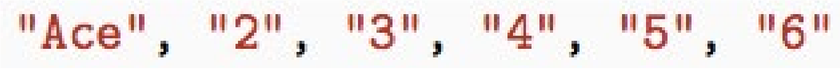
Card



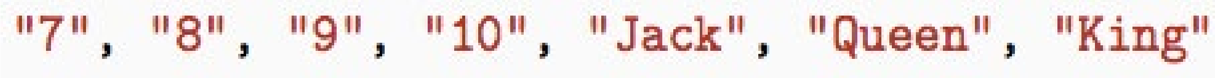
```
public class Card {

    private int suit, rank;

    private String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};

    public Card (int s, int r)  {
        this.suit = s;  this.rank = r;
    }

    public int getSuit(){
        return suit;
    }

    public int getRank(){
        return rank;
    }

    @Override
    public String toString(){
        return suits[suit]; // this returns the suit value but not the rank
    }
}
```

rank

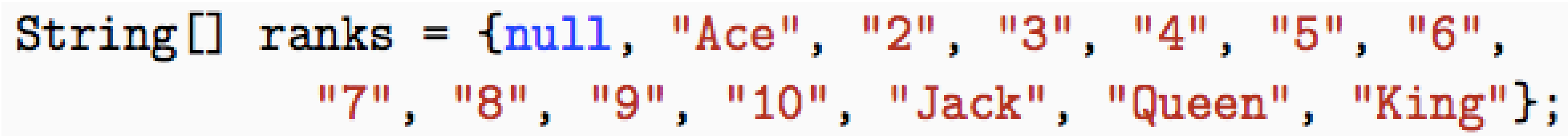
- We can do something similar to hold the possible 13 values of the rank of a Card



```
"Ace", "2", "3", "4", "5", "6"
```

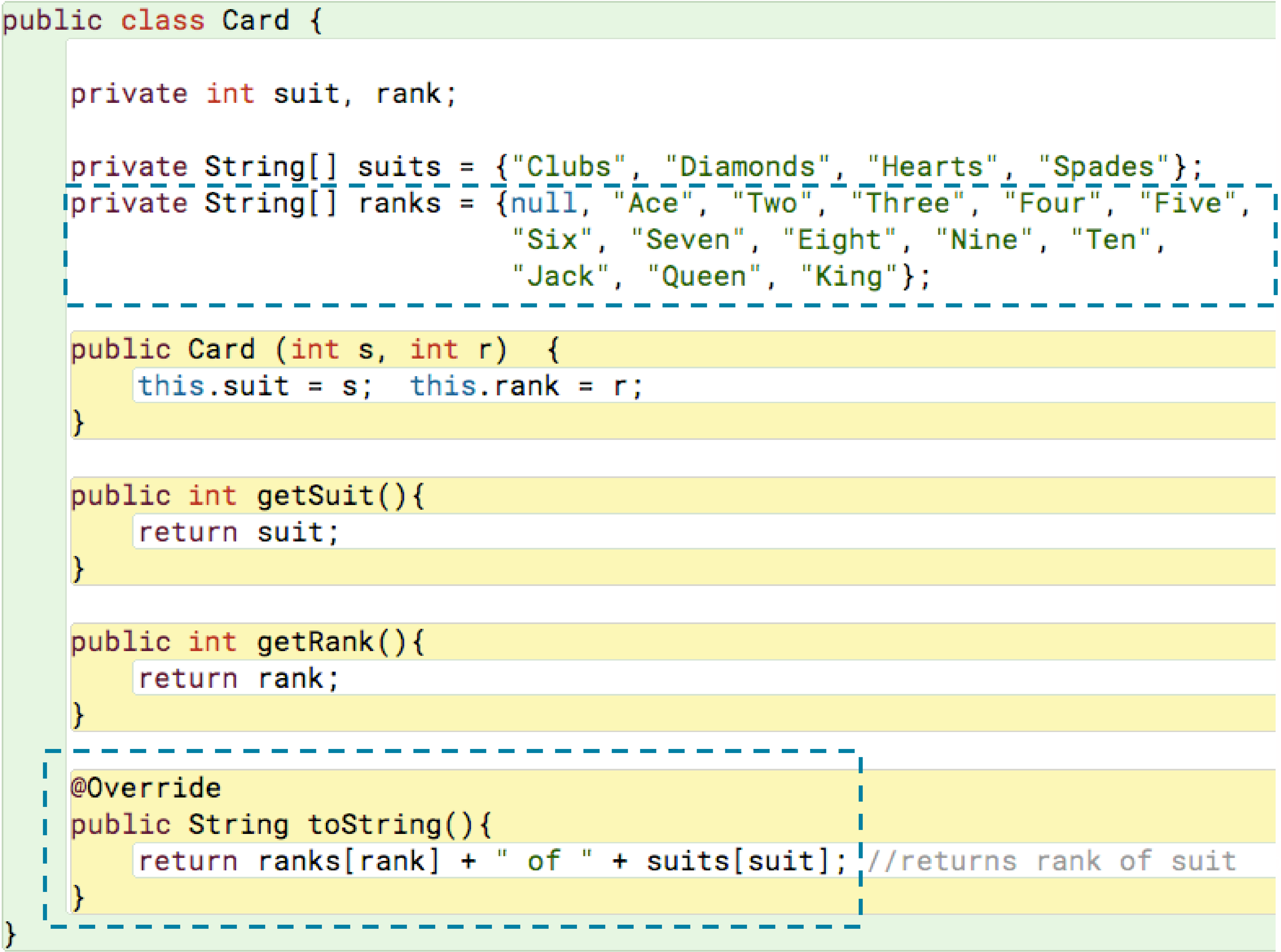


```
"7", "8", "9", "10", "Jack", "Queen", "King"
```

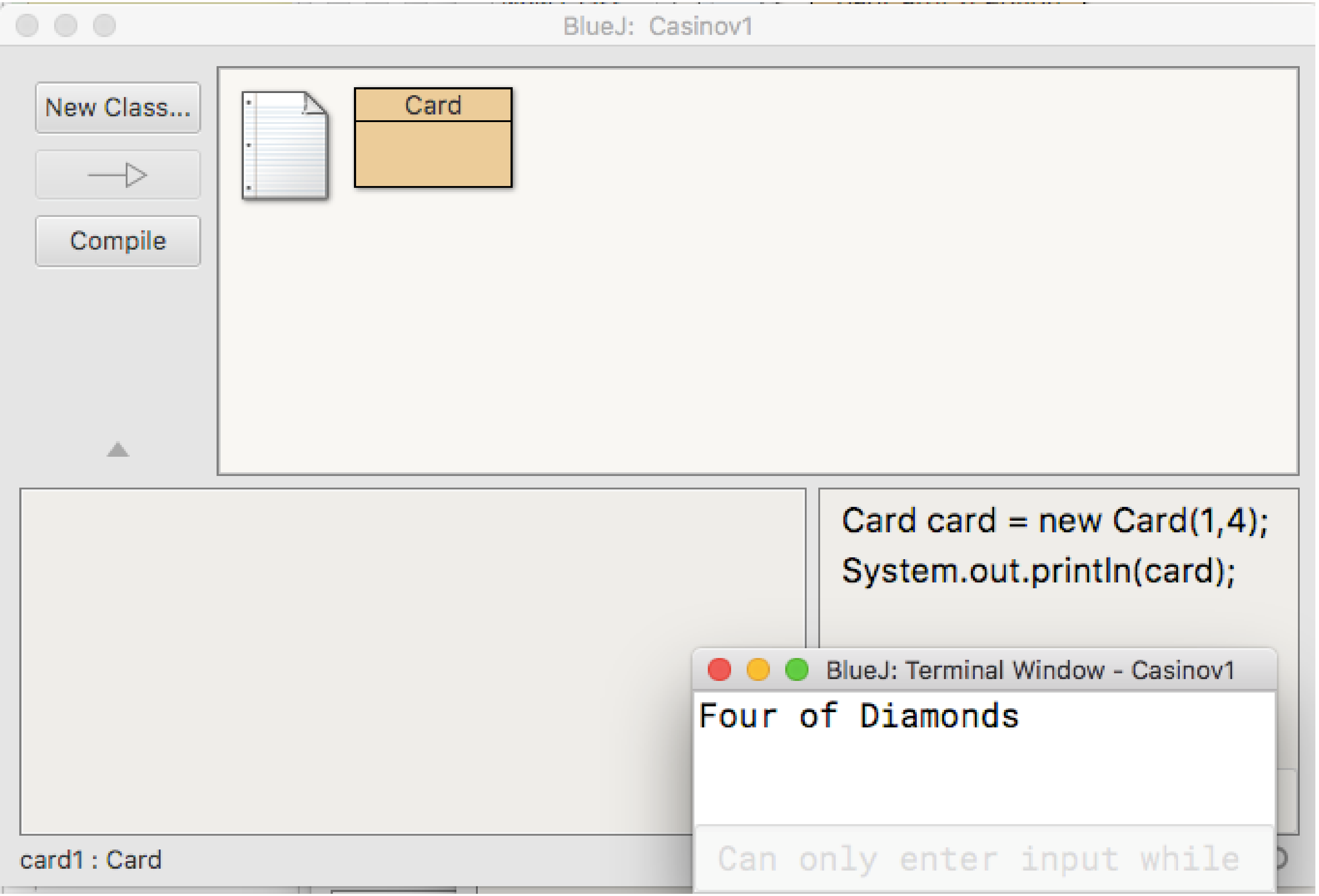


```
String[] ranks = {null, "Ace", "2", "3", "4", "5", "6",
                    "7", "8", "9", "10", "Jack", "Queen", "King"};
```

Q: Why is null the first value in the RANKS array?



```java
public class Card {

    private int suit, rank;

    private String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
    private String[] ranks = {null, "Ace", "Two", "Three", "Four", "Five",
                              "Six", "Seven", "Eight", "Nine", "Ten",
                              "Jack", "Queen", "King"};

    public Card (int s, int r)  {
        this.suit = s;  this.rank = r;
    }

    public int getSuit(){
        return suit;
    }

    public int getRank(){
        return rank;
    }

    @Override
    public String toString(){
        return ranks[rank] + " of " + suits[suit]; //returns rank of suit
    }
}
```



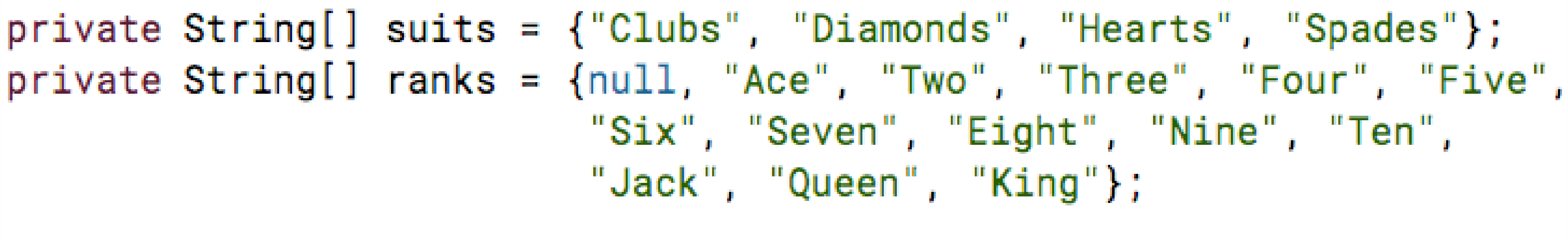
```
Card card = new Card(1,4);
System.out.println(card);
```

```
Four of Diamonds
```

Blackboard

- Download the Card Code from Blackboard, Week 11.
- Add it to a project in BlueJ

Introducing static fields



```
private String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
private String[] ranks = {null, "Ace", "Two", "Three", "Four", "Five",
                          "Six", "Seven", "Eight", "Nine", "Ten",
                          "Jack", "Queen", "King"};
```

- Suits and ranks arrays are declared in every object of type Card
- This is wasteful (in terms of memory) and redundant (bad programming practice)
- The suits and ranks values are constant. They never change. They are the same for every Card object
- In situations like this, you should declare these variables to be **static**

Static fields

- Up to now, the instance variables/fields you have used have scope at object level
- A static field is a variable that exists and has scope at **class** level
- Typically, it is used to hold constant, *non-changing* values
- Often, they may be declared public **and final.**
- This means that they can be accessed directly by other classes and objects but *cannot* be changed

Static fields

- Generally, Static variables are capitalised
- Generally declared as **public**
- Very often declared as **final**

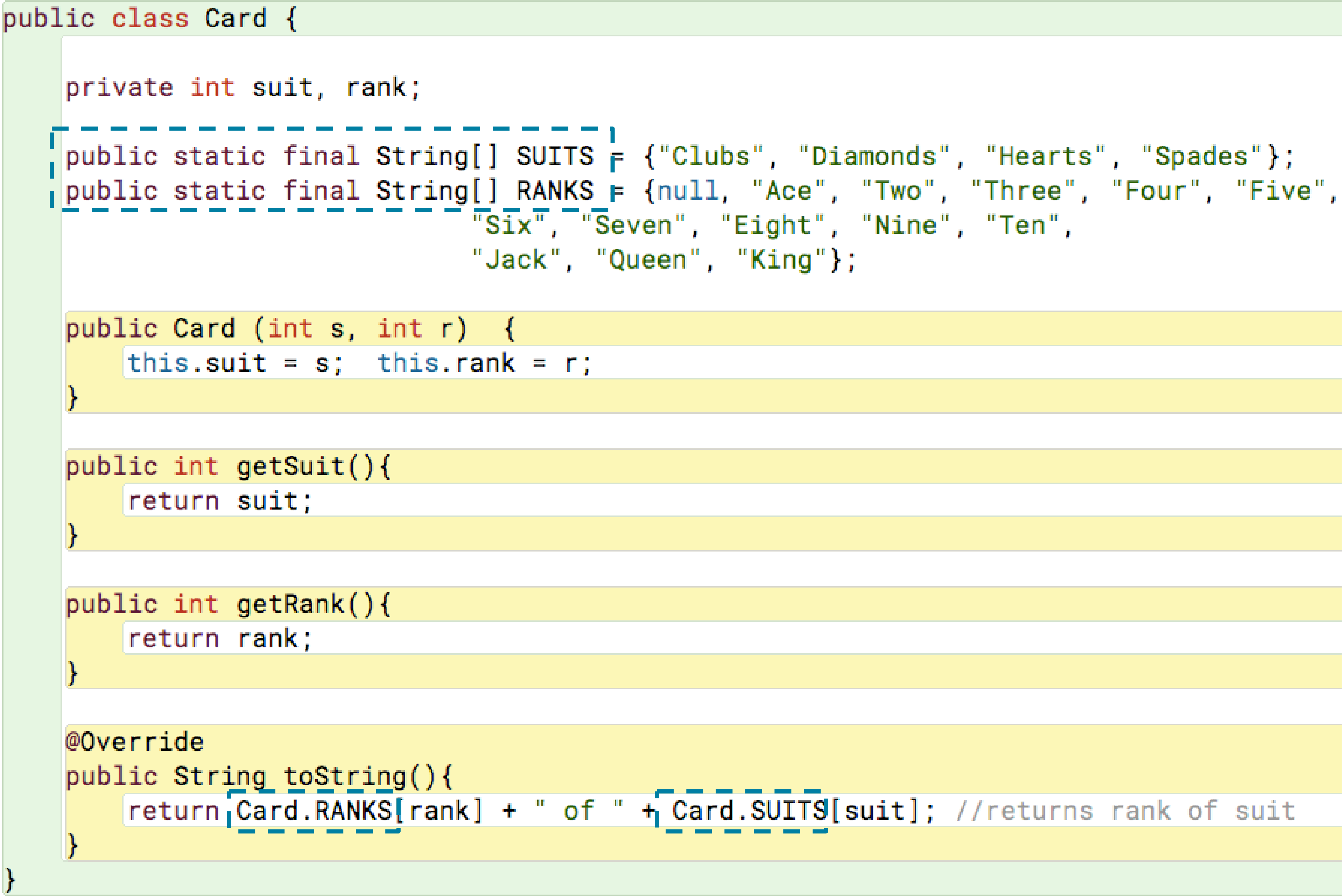
- You use them when you want to declare a value/property that is **unchanging or common to all objects of a class**

Static fields

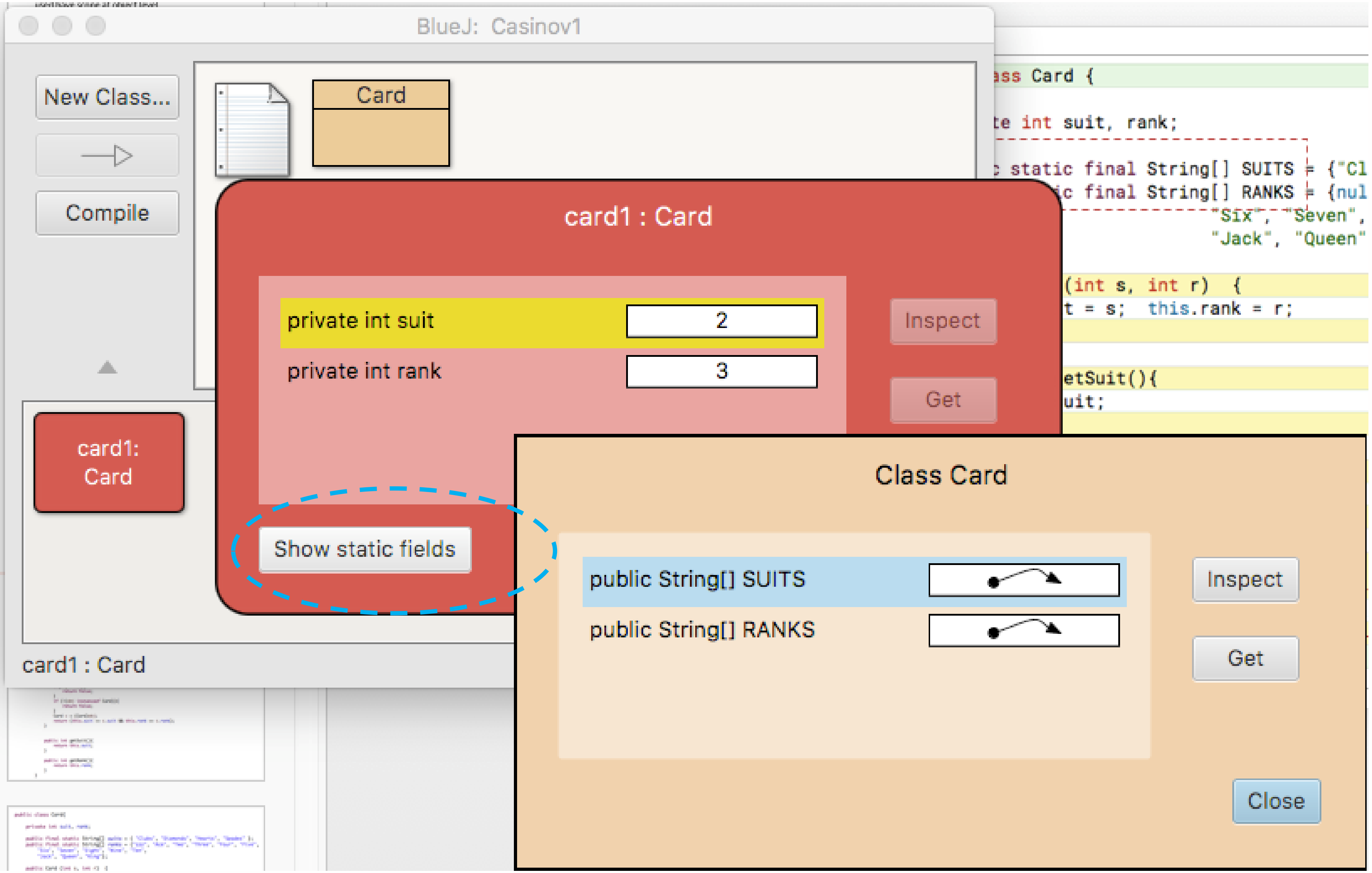
When referring to a static field, use the form ClassName.STATIC_VARIABLE_NAME

E.g

```
Card.RANKS
Card.SUITS
Math.PI
```



```
public class Card {

    private int suit, rank;

    public static final String[] SUITS = {"Clubs", "Diamonds", "Hearts", "Spades"};
    public static final String[] RANKS = {null, "Ace", "Two", "Three", "Four", "Five",
                                 "Six", "Seven", "Eight", "Nine", "Ten",
                                 "Jack", "Queen", "King"};

    public Card (int s, int r)  {
        this.suit = s;  this.rank = r;
    }

    public int getSuit(){
        return suit;
    }

    public int getRank(){
        return rank;
    }

    @Override
    public String toString(){
        return Card.RANKS[rank] + " of " + Card.SUITS[suit]; //returns rank of suit
    }
}
```



```
ass Card {
te int suit, rank;
: static final String[] SUITS = {"Cl
ic final String[] RANKS = {nul
                              "Six", "Seven",
                              "Jack", "Queen"
(int s, int r)  {
t = s;  this.rank = r;
etSuit(){
uit;
```

card1 : Card

private int suit	2
private int rank	3

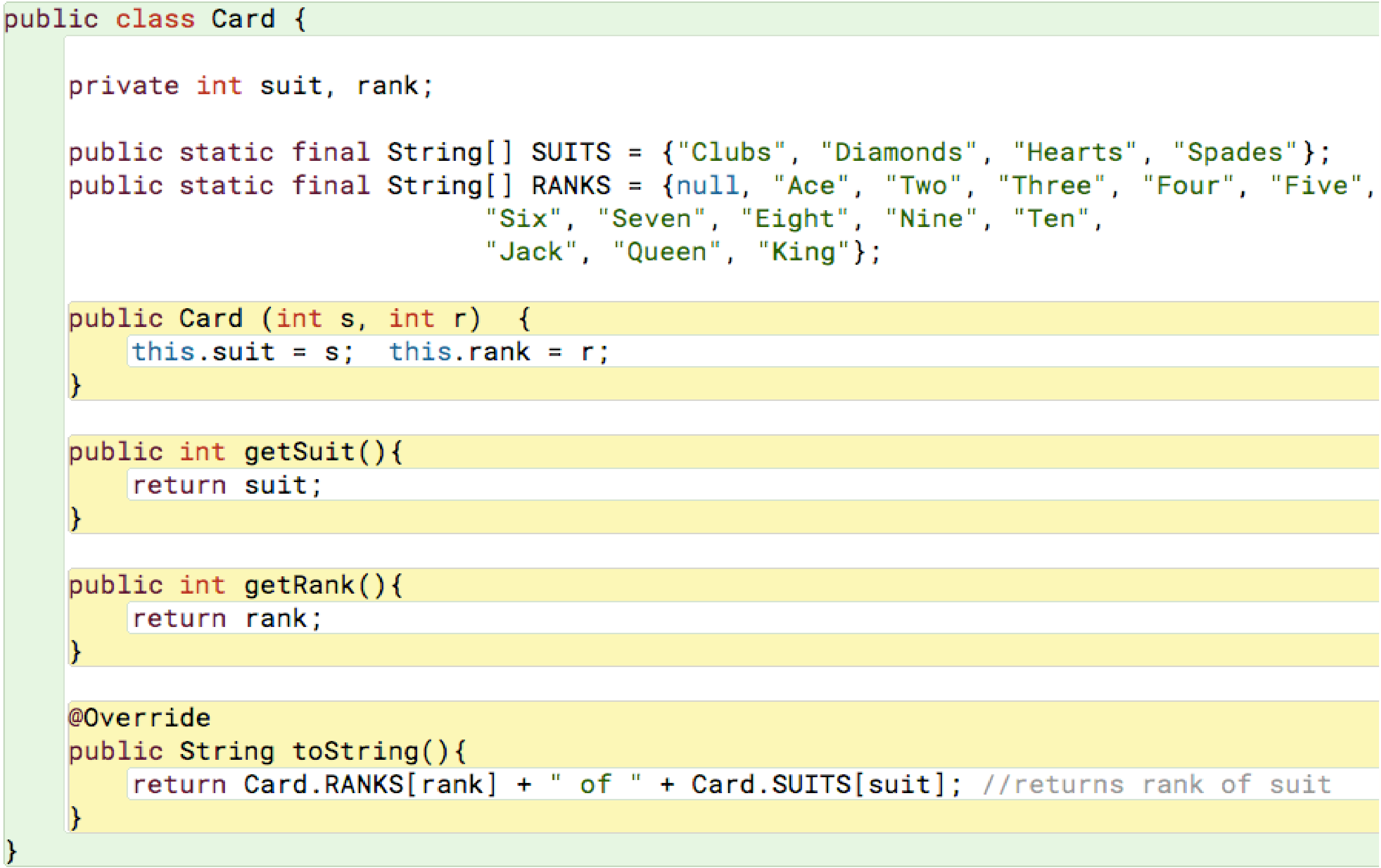
Class Card

public String[] SUITS	●→
public String[] RANKS	●→

Exception Handling

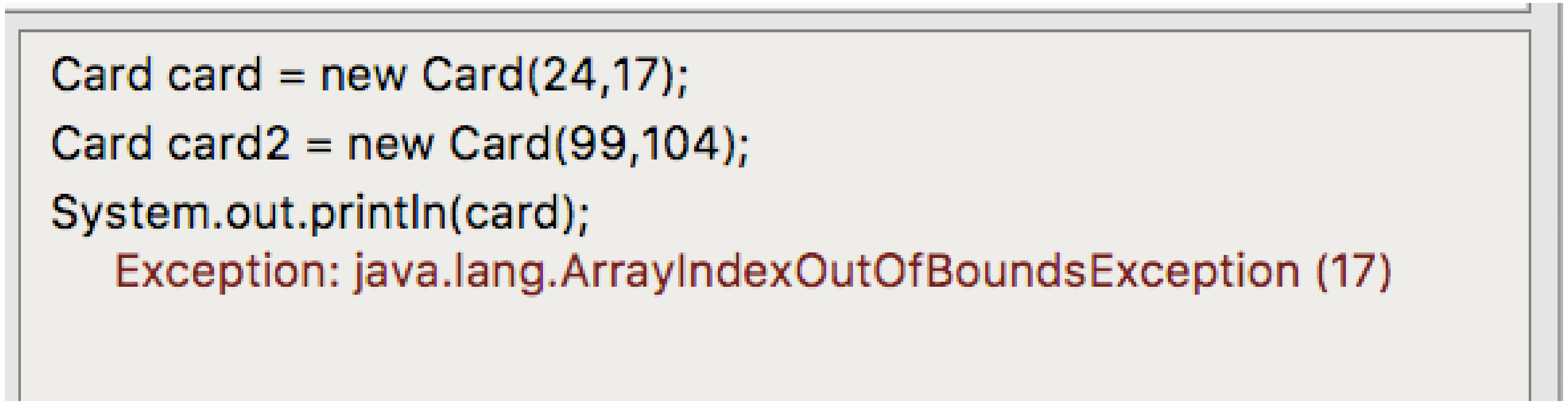
Card Class

Our Card class has a significant weakness

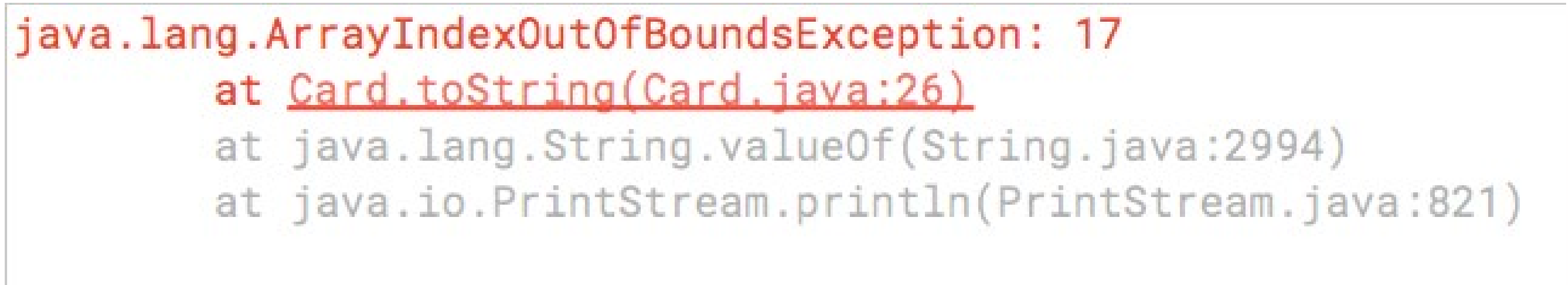


```
public class Card {

    private int suit, rank;

    public static final String[] SUITS = {"Clubs", "Diamonds", "Hearts", "Spades"};
    public static final String[] RANKS = {null, "Ace", "Two", "Three", "Four", "Five",
                                    "Six", "Seven", "Eight", "Nine", "Ten",
                                    "Jack", "Queen", "King"};

    public Card (int s, int r)  {
        this.suit = s;  this.rank = r;
    }

    public int getSuit(){
        return suit;
    }

    public int getRank(){
        return rank;
    }

    @Override
    public String toString(){
        return Card.RANKS[rank] + " of " + Card.SUITS[suit]; //returns rank of suit
    }
}
```

Handling invalid values



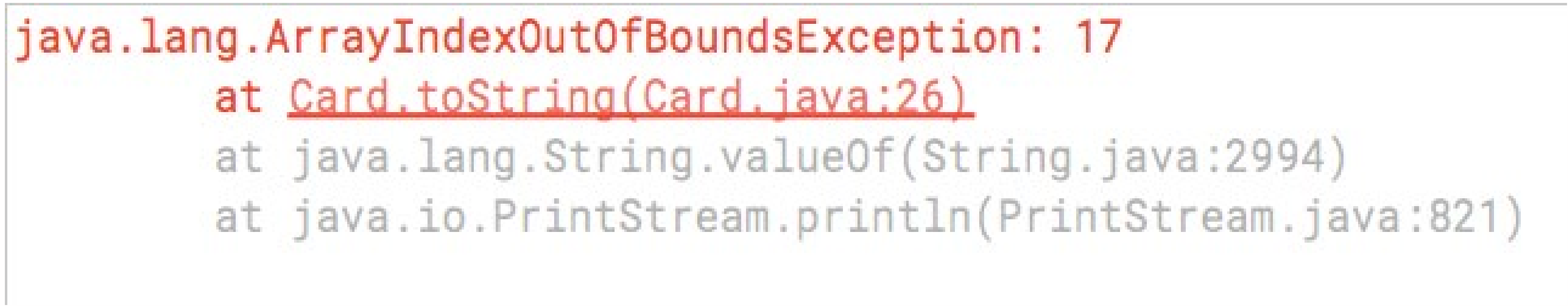
```
Card card = new Card(24,17);
Card card2 = new Card(99,104);
System.out.println(card);
    Exception: java.lang.ArrayIndexOutOfBoundsException (17)
```



```
java.lang.ArrayIndexOutOfBoundsException: 17
	at Card.toString(Card.java:26)
	at java.lang.String.valueOf(String.java:2994)
	at java.io.PrintStream.println(PrintStream.java:821)
```

- It allows us to create Card objects with invalid Card values.
- The error will only be detected later in the program

An Exception



```
java.lang.ArrayIndexOutOfBoundsException: 17
        at Card.toString(Card.java:26)
        at java.lang.String.valueOf(String.java:2994)
        at java.io.PrintStream.println(PrintStream.java:821)
```

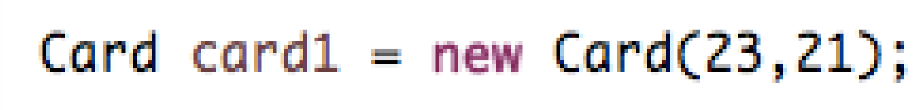
- The error message above is from the Java Runtime Environment (JRE)
- It tells use that an Exception was generated and was not handled
- This has caused the program to crash

What is an Exception?

- An exception is an “*exceptional event*” – one that may lead to a serious error in your program if not handled appropriately.
- An exception is generated only when the program runs – hence it is known as a **runtime error**
- Very often, the error (and the exception generated) occurs **when the program is asked to do something that is impossible for it to do**
- In Java, each exception is represented by an **Exception object**

Programming for Exceptions

- As the programmer, it your responsibility to anticipate the situations in which your program will fail
- You have to write code to manage any **exceptional events** that may occur within your program
- In our example, an exceptional event is when a user tries to get our program to instantiate an invalid card

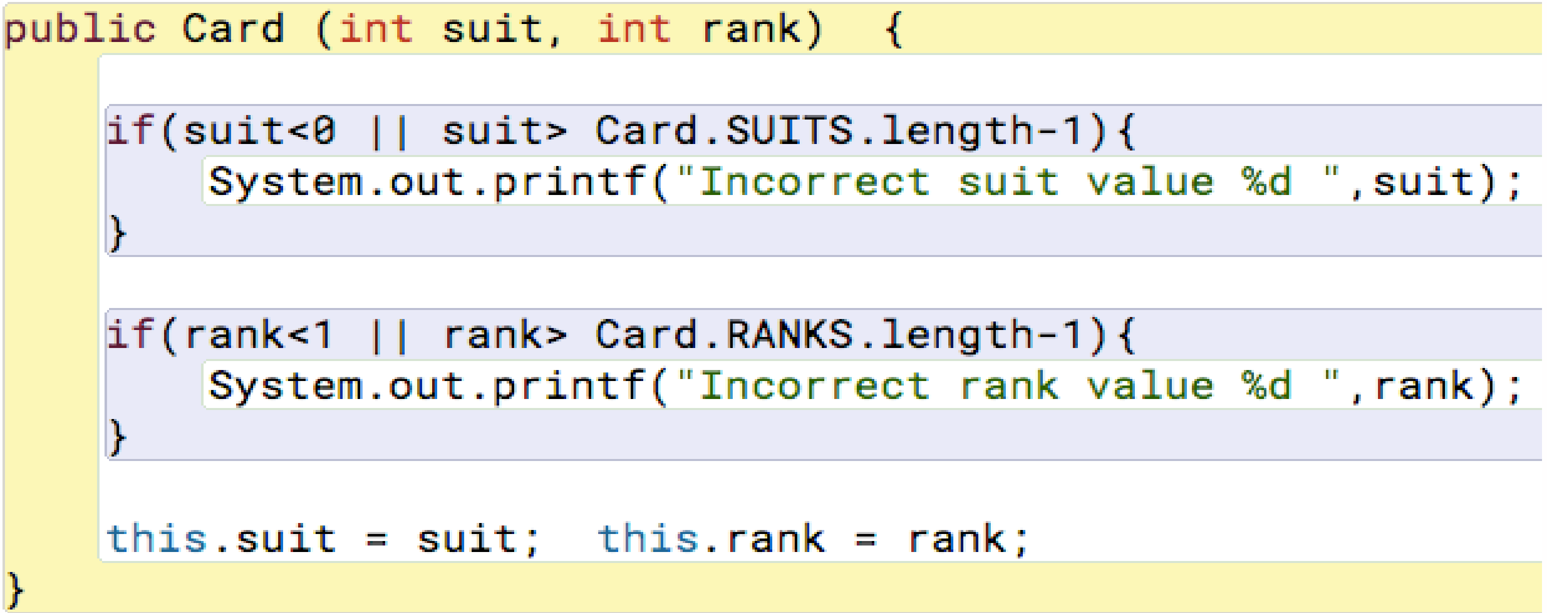


```
Card card1 = new Card(23,21);
```

- If this card object gets into say, a poker program, it will wreak havoc, as all other objects will expect Card objects with valid suit and rank values

Checking valid input for a Card

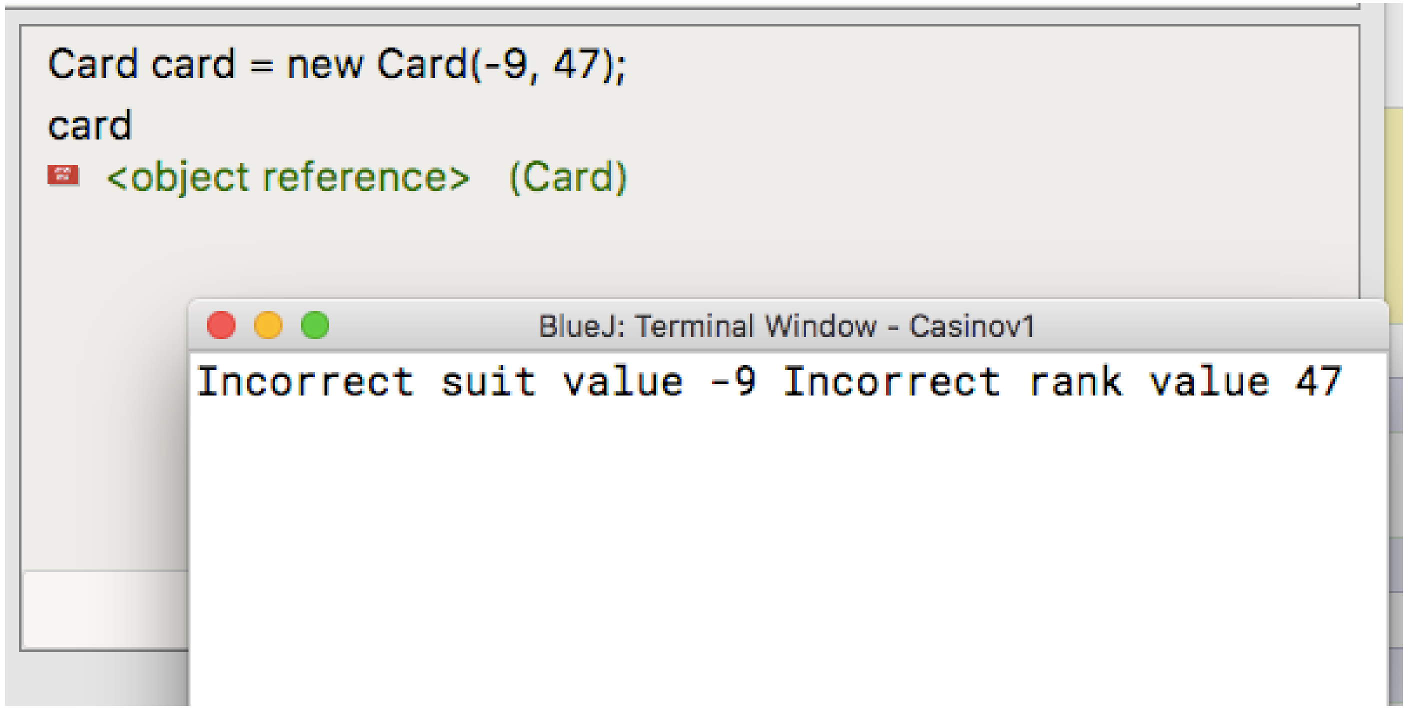
- The key question is how to programmatically handle the situation when invalid input is entered.
- In the case of the Card, we might write the following in the constructor:



```
public Card (int suit, int rank)  {

    if(suit<0 || suit> Card.SUITS.length-1){
        System.out.printf("Incorrect suit value %d ",suit);
    }

    if(rank<1 || rank> Card.RANKS.length-1){
        System.out.printf("Incorrect rank value %d ",rank);
    }

    this.suit = suit;  this.rank = rank;
}
```

Weak approach

- It prints out a warning message only
- The invalid Card object is still created



```
Card card = new Card(-9, 47);
card
<object reference>   (Card)
```

```
Incorrect suit value -9 Incorrect rank value 47
```

Detect error-> Throw an Exception

- We want an approach that **prevents an invalid object being created**
- Java has the concept of an Exception object that can be created to stop a program going any further
- When a program generates an Exception object it is said to **throw an Exception**
- When an Exception is thrown, the program must have code in place to **catch it**
- **If not, the program terminates**

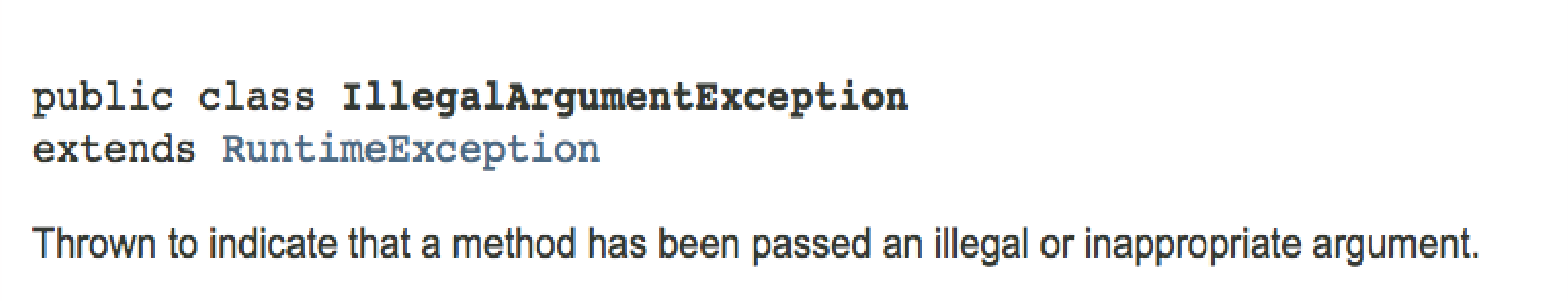
Throwing an Exception

This involves

1. Detecting an error
2. Creating an Exception object
3. Passing the Exception object to The Java Runtime Environment (JRE) Exception Handling Procedures. This also means the execution of the method does not complete
4. The JRE then looks for part of your program to take responsibility for this error.
5. In other words, your program should also have code ready to **catch** the error

Card throws IllegalArgumentException

In our case, we can make the Card throw an Exception - an **IllegalArgumentException**

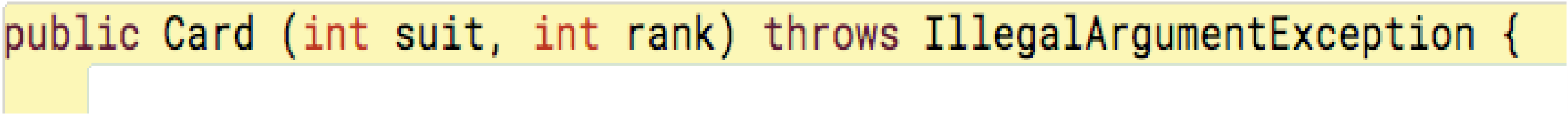


```
public class IllegalArgumentException
extends RuntimeException
```

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

throws

When you want a method to throw an Exception you add throws and the Exception type to the method signature



```
public Card (int suit, int rank) throws IllegalArgumentException {
```

This tells any code that wants to call the constructor method that it may throw an **IllegalArgumentException**

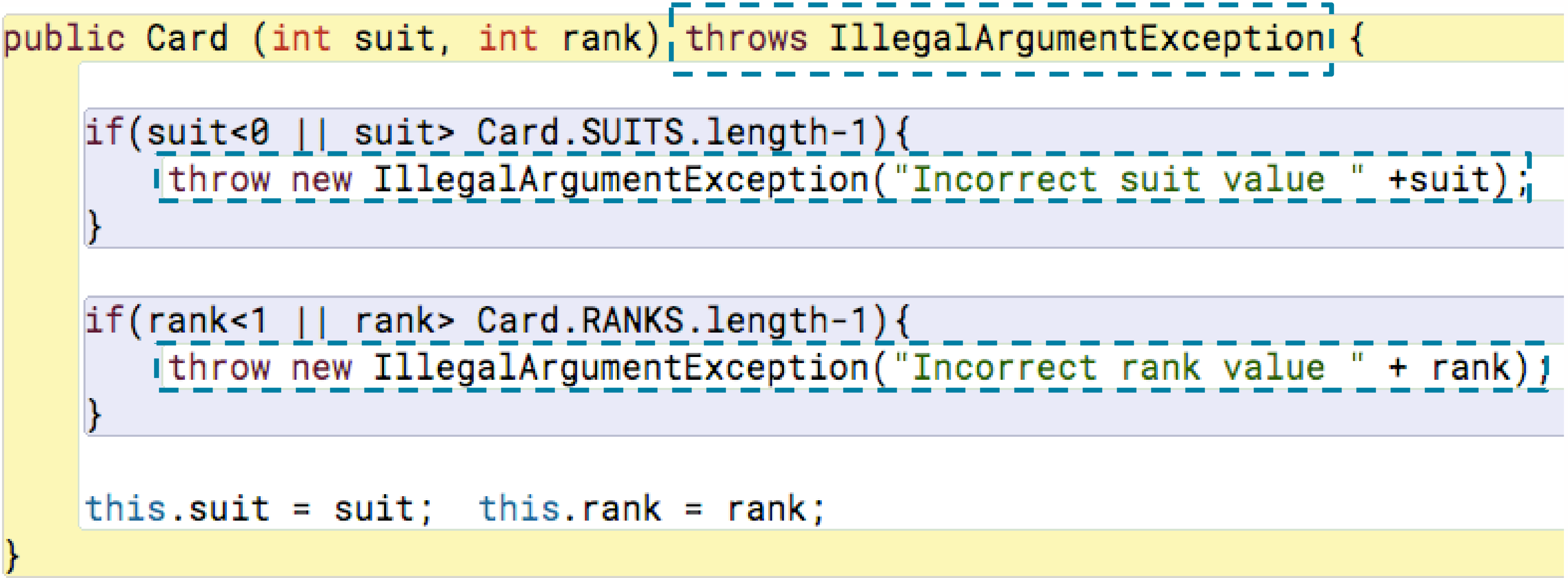
It will be up to the calling code to handle that exception if it is thrown

throws

- The Card constructor has to define conditions which will cause it to throw an Exception.
- These are the same conditions that caused it to issue a weak warning
- Instead now, it generates and **throws** a new Exception object
- To throw an Exception you use the throw keyword

Revised Card constructor

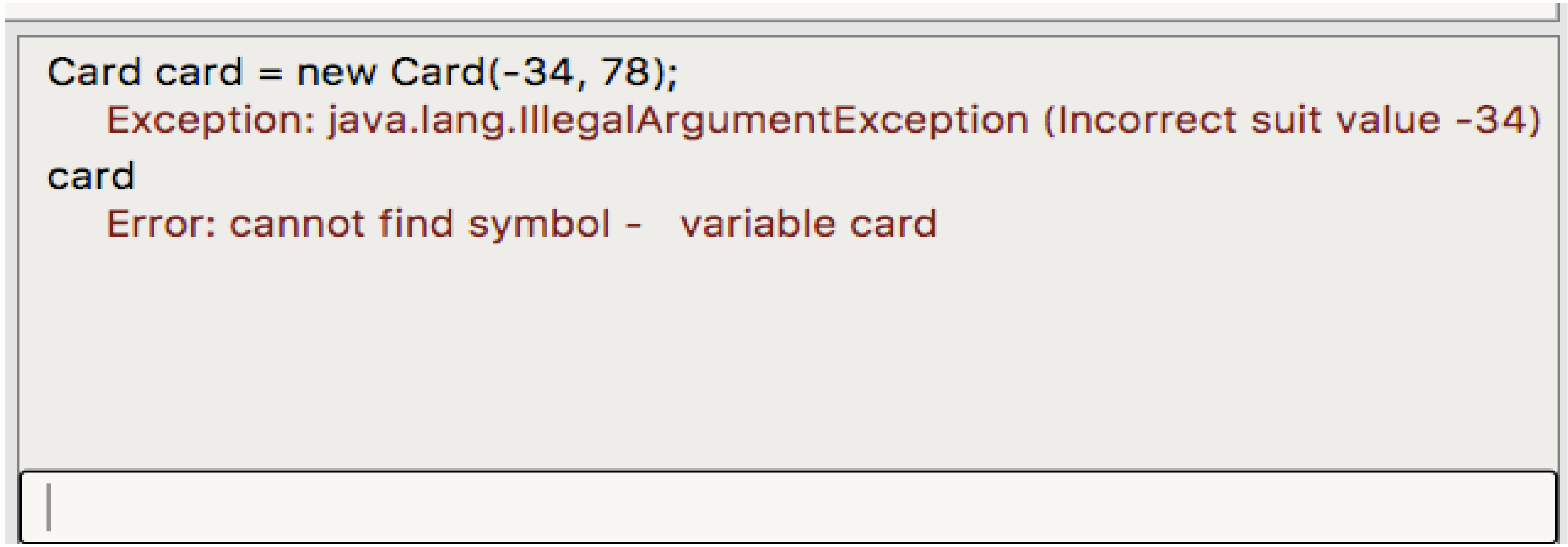
- When an Exception is thrown, execution of the method stops
- As this is a *constructor* method, this means that the (invalid) Card object is not created



```
public Card (int suit, int rank) throws IllegalArgumentException {

    if(suit<0 || suit> Card.SUITS.length-1){
        throw new IllegalArgumentException("Incorrect suit value " +suit);
    }

    if(rank<1 || rank> Card.RANKS.length-1){
        throw new IllegalArgumentException("Incorrect rank value " + rank);
    }

    this.suit = suit;  this.rank = rank;
}
```

Testing out code

- Now when we try to create a Card with invalid values, we will fail.
- An exception is thrown.
- The card variable below is not assigned to a Card object



```
Card card = new Card(-34, 78);
    Exception: java.lang.IllegalArgumentException (Incorrect suit value -34)
card
    Error: cannot find symbol -   variable card
```

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

throwing and catching

- If your method **throws** an exception

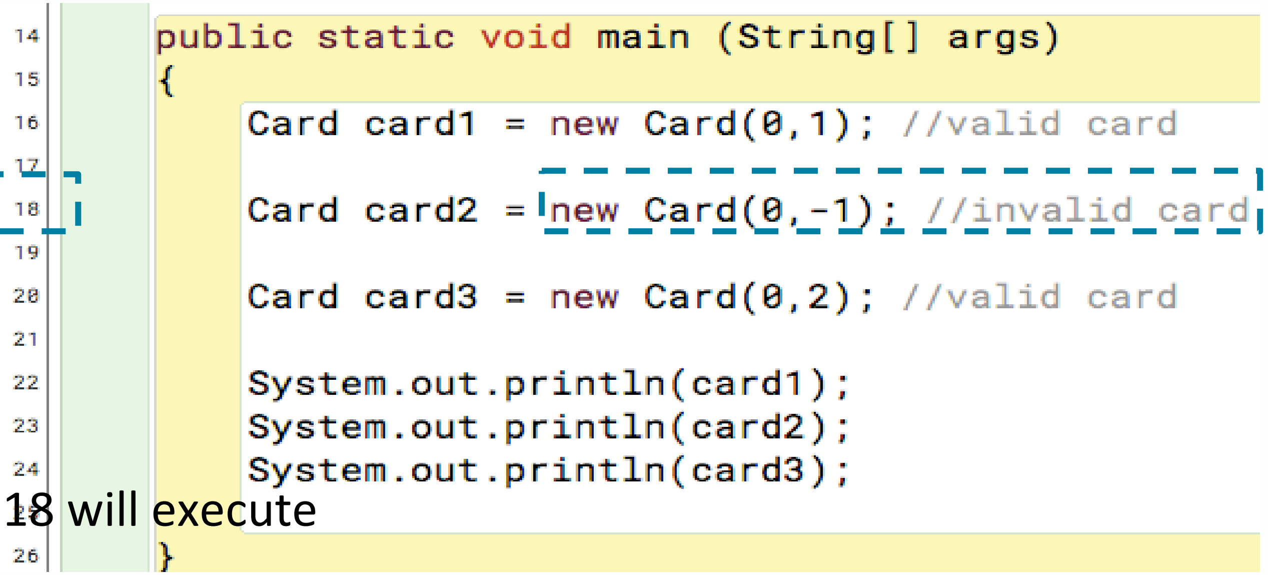
- Then you **must** also have code in place to **catch** and **handle** the exception

Graceful recovery

- If an exception is not caught, the JRE will terminate the program
- This is a drastic step
- In most cases, you will want your program to recover (gracefully) from an exception and carry on
- This involves **catching** the Exception that has been generated

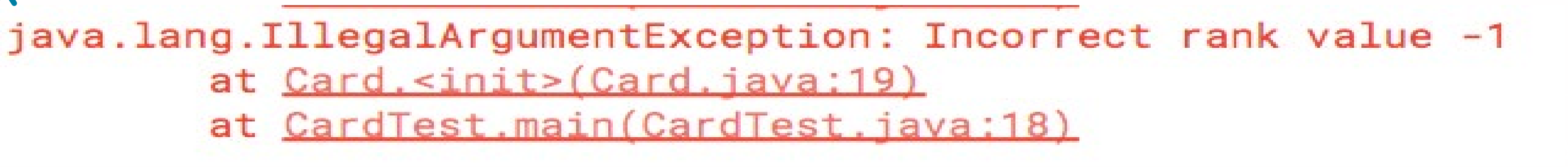
Example of program termination

- If you run the following code, the **uncaught exception** will terminate the program at line 18



```
14  public static void main (String[] args)
15  {
16      Card card1 = new Card(0,1); //valid card
17
18      Card card2 = new Card(0,-1); //invalid card
19
20      Card card3 = new Card(0,2); //valid card
21
22      System.out.println(card1);
23      System.out.println(card2);
24      System.out.println(card3);
25
26  }
```

- Nothing after line 18 will execute



```
java.lang.IllegalArgumentException: Incorrect rank value -1
        at Card.<init>(Card.java:19)
        at CardTest.main(CardTest.java:18)
```

Try/catch

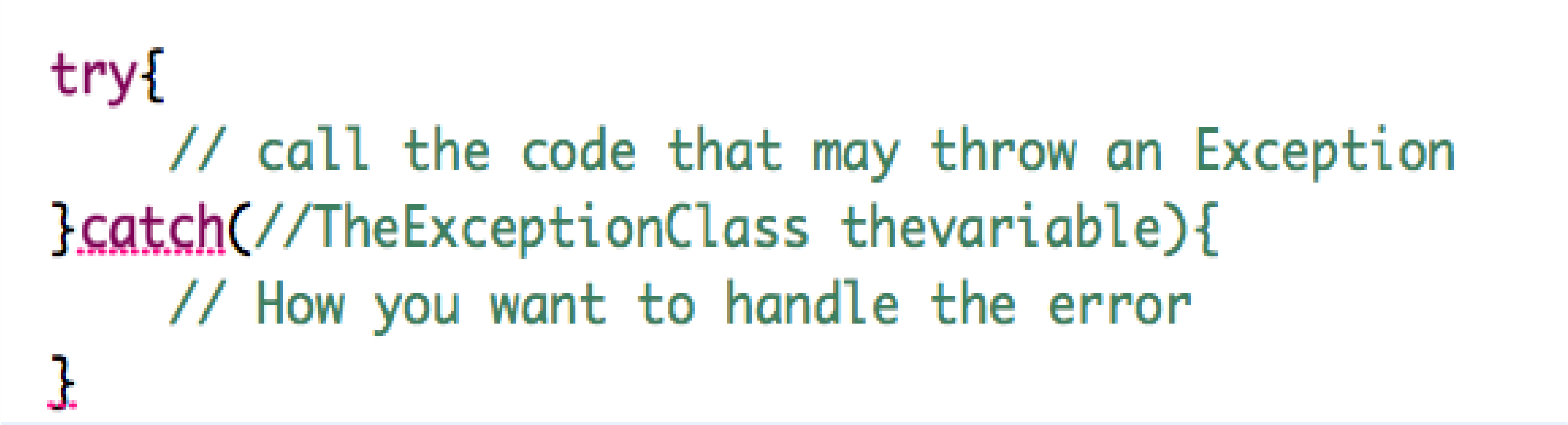
- If you want the program to recover from the Exception, you have to catch and handle it
- This means using a try/catch expression

- **Try:** try to execute this piece of code. If it executes without throwing an exception. Fine. There is no need to for **the catch** clause to be executed
- **Catch:** if an exception has been thrown then execute this piece of recovery code to **handle** the Exception (very often just an error message)

General format of try/catch block

Meaning:

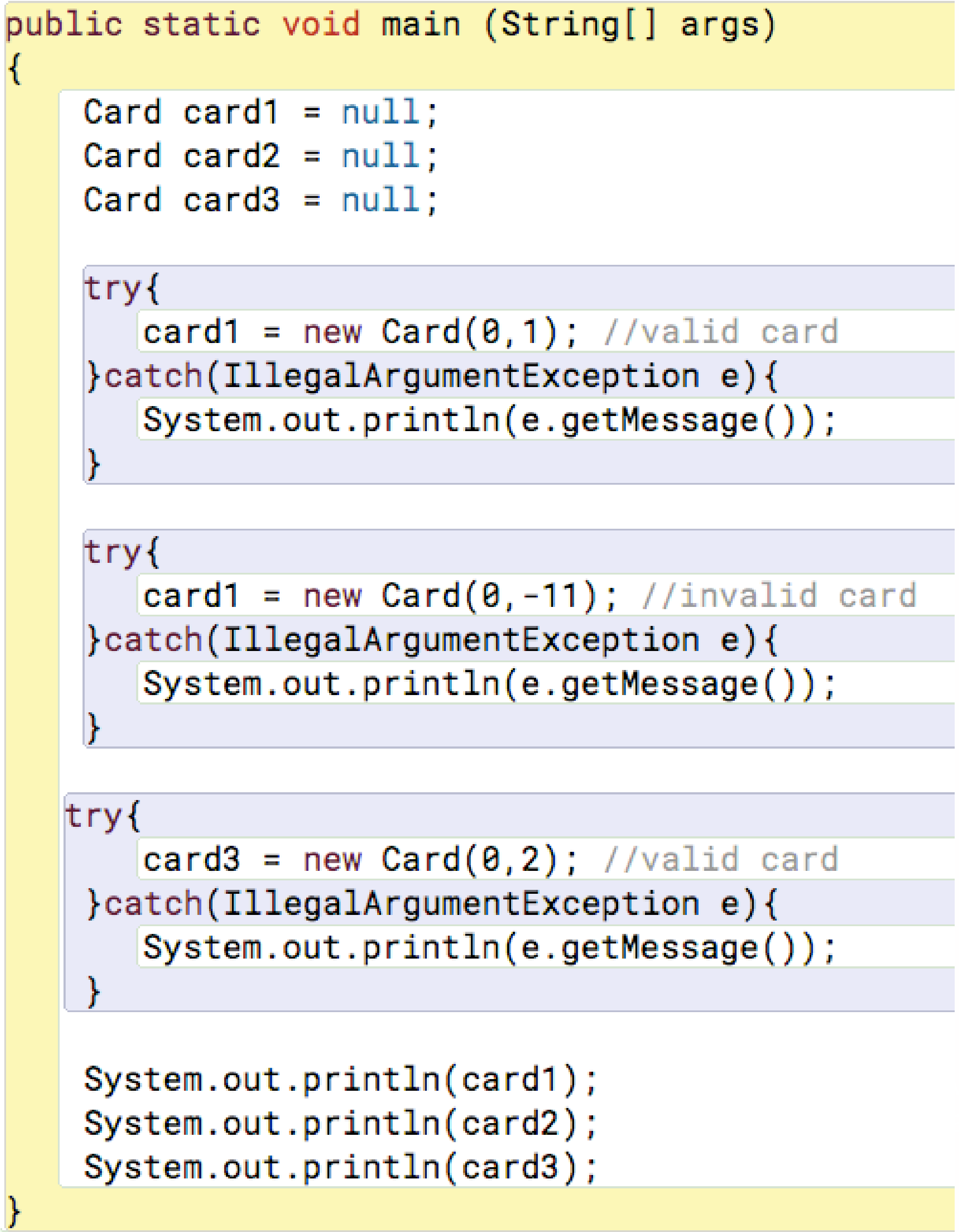
1. Try to call this method (which may throw an Exception)
2. If it throws an exception object, catch it! (the exception will go no further)
3. Then handle the exception this way
4. Carry on to the next line of execution (as normal)



```
try{
    // call the code that may throw an Exception
}catch(//TheExceptionClass thevariable){
    // How you want to handle the error
}
```

Revised Example

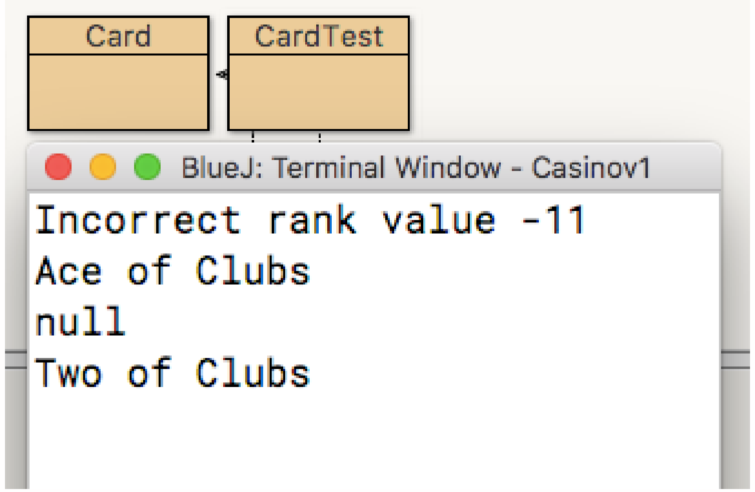
- Each call to the Card constructor is wrapped in a try/catch block
- If an Exception is thrown, it will be caught and handled
- This allows the program to execute until the end.



```
public static void main (String[] args)
{
    Card card1 = null;
    Card card2 = null;
    Card card3 = null;

    try{
        card1 = new Card(0,1); //valid card
    }catch(IllegalArgumentException e){
        System.out.println(e.getMessage());
    }

    try{
        card1 = new Card(0,-11); //invalid card
    }catch(IllegalArgumentException e){
        System.out.println(e.getMessage());
    }

    try{
        card3 = new Card(0,2); //valid card
    }catch(IllegalArgumentException e){
        System.out.println(e.getMessage());
    }

    System.out.println(card1);
    System.out.println(card2);
    System.out.println(card3);
}
```

Graceful recovery

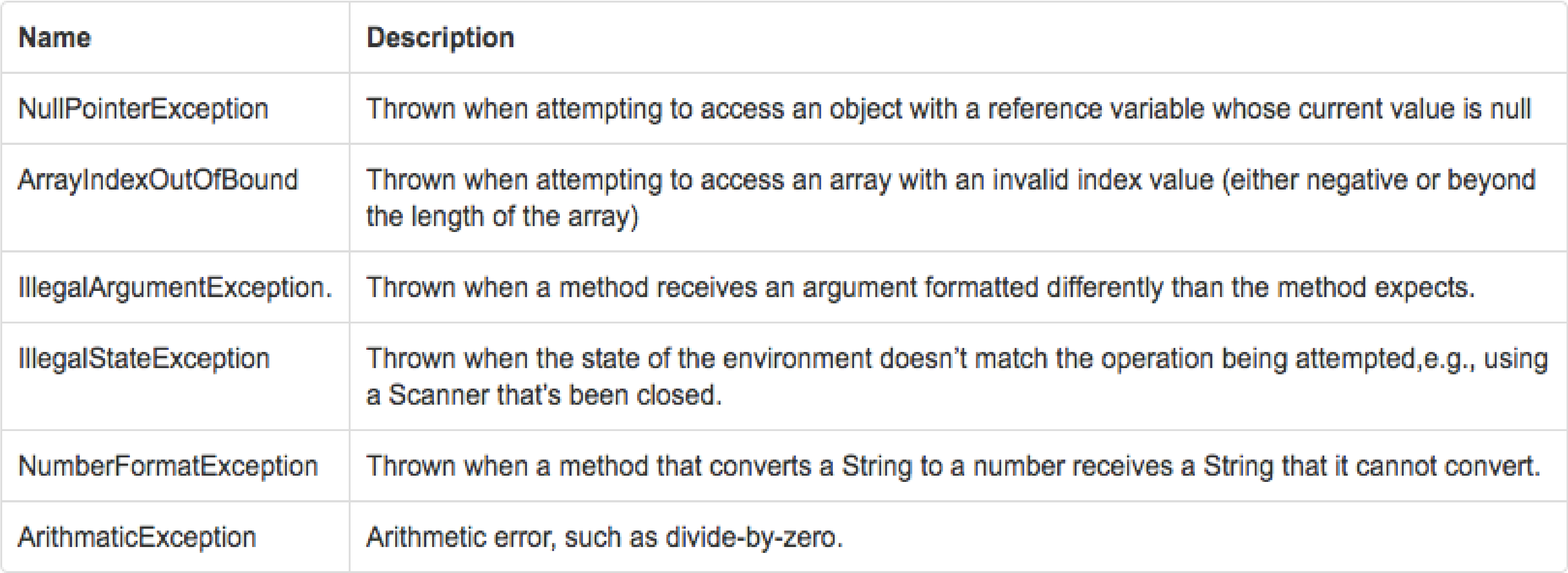
- Now when we run the program we get an error message caused by the attempt to create the invalid second Card



```
Incorrect rank value -11
Ace of Clubs
null
Two of Clubs
```

- The invalid second card is not created
- **The program can continue** on to create the third card (“Two of Clubs)
- It then prints out the values of the card1, card2 and card3 variables
- (card2 is pointing to null, because the second invalid card was not created)

Some common unchecked Exceptions



Name	Description
NullPointerException	Thrown when attempting to access an object with a reference variable whose current value is null
ArrayIndexOutOfBound	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array)
IllegalArgumentException.	Thrown when a method receives an argument formatted differently than the method expects.
IllegalStateException	Thrown when the state of the environment doesn't match the operation being attempted,e.g., using a Scanner that's been closed.
NumberFormatException	Thrown when a method that converts a String to a number receives a String that it cannot convert.
ArithmaticException	Arithmetic error, such as divide-by-zero.

Wrapping up

- A static field is a variable that exists and has scope at **class** level
- You use them when you want to declare a value/property that is common to all objects of a class
- You can anticipate when errors may be generated by your program and write exceptions throwing code to cover these events
- You also have to write code to catch and handle **exceptions** that may occur within your program