UNIVERSITY of VIRGINIA

CS 2100: Data Structures & Algorithms 1

Classes and Enums (& How to use the Java API)

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Classes

- What does “ `public class ______` ” actually mean?

Classes

- Classes define **objects**, the building blocks (or *blueprints*) of your program
- A **class** describes a **data type**!
  - It lists a set of attributes (fields), and actions/behaviors (methods)
    - **State**
      - Variables, fields
      - The values of the fields describe the state of each object in the class
    - **Behavior**
      - Methods
      - What can this object do – the behavior of each object in the class

Classes

- **Fields** and **methods** may be:
  - **Static**: available without an instance of the class
    - Using `static` keyword
  - **Instance**: called only as a method on an object in memory

- **Methods** may be:
  - **Accessor**: read the state of the object in memory
  - **Mutator**: change the state of the object in memory

- Variables can be (and usually are) other class objects!

- Classes provide ***encapsulation***

Known as "***getters***" and "***setters***" e.g. methods `getName()` and `setName(String newName)`

Card / Deck Classes

Discussing Parts of a Class, including:

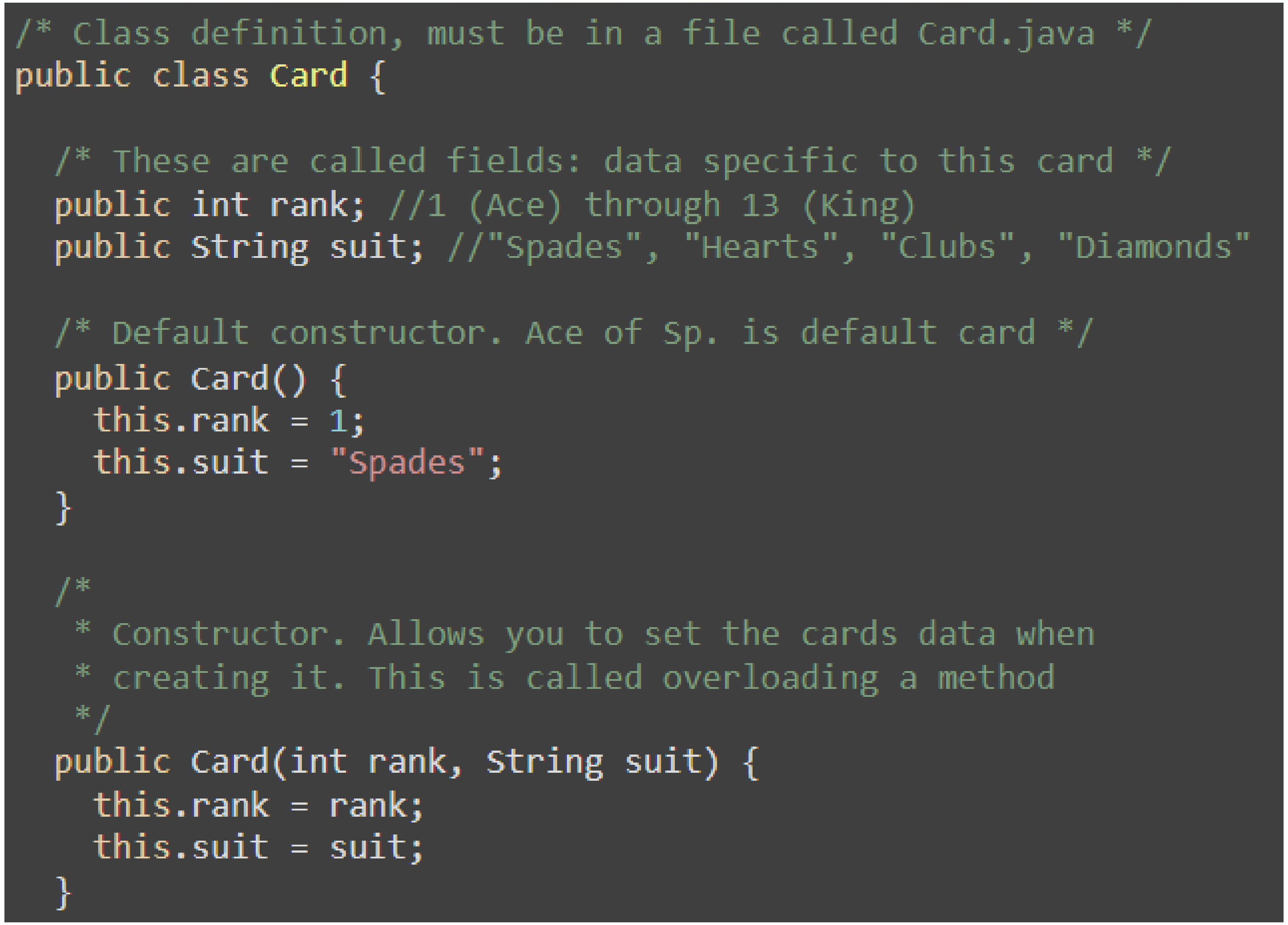
Constructors / Constructor Overloading

toString() and Getters/Setters

Writing Classes

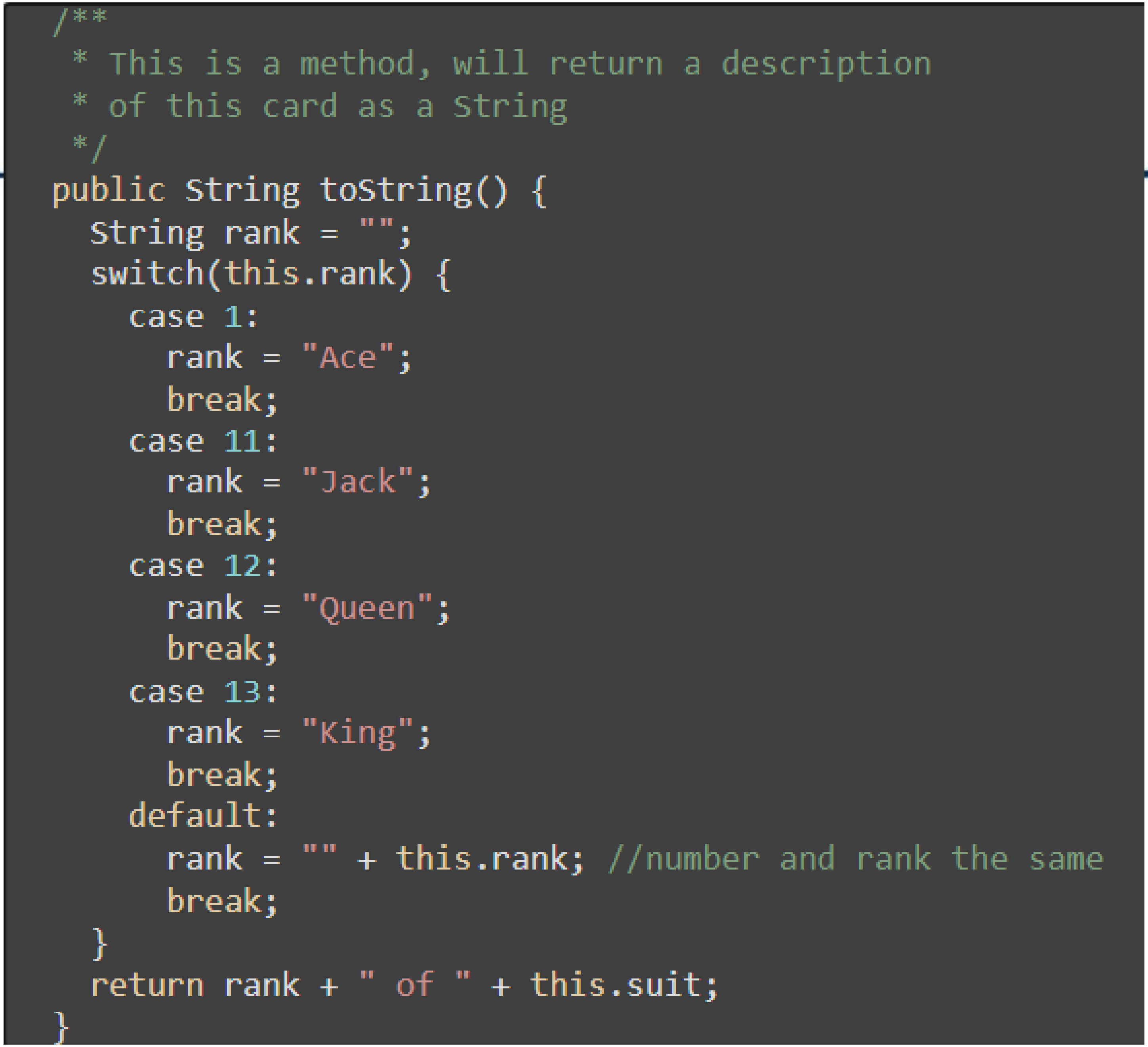
- For example…
  - Suppose I’d like to write code for playing card games.
  - It would be nice to have variables for type Card, Deck, etc. to make this easier
  - Card, Deck,etc. are *not* native / built-in to Java. We have to create them ourselves.
  - Creating classes allows us to create our own **Objects** (that is, **Data Types**!)

Example Class: Playing Cards (Card Class / Card.java)



```java
/* Class definition, must be in a file called Card.java */
public class Card {

  /* These are called fields: data specific to this card */
  public int rank; //1 (Ace) through 13 (King)
  public String suit; //"Spades", "Hearts", "Clubs", "Diamonds"

  /* Default constructor. Ace of Sp. is default card */
  public Card() {
    this.rank = 1;
    this.suit = "Spades";
  }

  /*
   * Constructor. Allows you to set the cards data when
   * creating it. This is called overloading a method
   */
  public Card(int rank, String suit) {
    this.rank = rank;
    this.suit = suit;
  }
```

Example Class: Playing Cards (Card Class / Card.java)



```
/**
 * This is a method, will return a description
 * of this card as a String
 */
public String toString() {
  String rank = "";
  switch(this.rank) {
    case 1:
      rank = "Ace";
      break;
    case 11:
      rank = "Jack";
      break;
    case 12:
      rank = "Queen";
      break;
    case 13:
      rank = "King";
      break;
    default:
      rank = "" + this.rank; //number and rank the same
      break;
  }
  return rank + " of " + this.suit;
}
```

Constructor

- A Java constructor is special method that is called when an object is *instantiated*. In other words, when you use the `new` keyword.
- A Java class constructor **initializes instances** (newly created objects) of that class.
- A constructor
  - creates space in memory (on the heap)
  - initializes all the **fields** of the object that need initialization (passed in as parameters)
  - sets the reference in the variable

Constructor

```
public class MyClass {

    private int number = 0;

    public MyClass() {

    }

    public MyClass(int theNumber) {

        this.number = theNumber;

    }

}
```

- The **first** part of a Java constructor declaration is an access modifier. *(Always “**public**” so can be used.)*
- The **second** part of a Java constructor declaration is the name of the class the constructor belongs to. Using the class name for the constructor signals to the Java compiler that this is a constructor. Also notice that **the constructor has no return type**, like other methods have.

Constructor

```
public class MyClass {

    private int number = 0;

    public MyClass() {

    }

    public MyClass(int theNumber) {

        this.number = theNumber;

    }

}
```

- The **third** part of a Java constructor declaration is a list of parameters the constructor can take. The constructor parameters are declared inside the parentheses () after the class name part of the constructor.
- The **fourth** part of a Java constructor declaration is the body of the constructor. The body of the constructor is defined inside the curly brackets **{ }** after the parameter list.

Constructor Overloading

```
public class MyClass {

    private int number = 0;

    public MyClass() {

    }

    public MyClass(int theNumber) {

        this.number = theNumber;

    }

}
```

- **Constructor overloading**
  - Multiple constructors in a Java class

- A class can have multiple constructors, as long as their signature (the parameters they take) are not the same. You can define as many constructors as you need (comes in multiple versions).

Another Constructor example (Employee)

```
public class Employee {
    public String firstName = null;
    public String lastName  = null;
    public int    birthYear = 0;

    // constructor:
    public Employee(String firstName, String lastName, int birthYear ) {

        this.firstName = firstName;
        this.lastName  = lastName;
        this.birthYear = birthYear;
    }

}
```

To signal to the Java compiler that you mean the fields of the Employee class (instance variables) and not the method parameters, put the **this** keyword and a dot in front of the field name. [CONVENTION]

How to print an Object? Use toString() method!

- The `toString()` method allows the programmer to specify how to print out an object.
- The `toString()` **method** returns the string representation of the object.
- String **return type** and takes in no parameters

- If you print any object, the **Java** compiler internally calls the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depends on your implementation.

  (from JavaPoint)

See code example:

```
// example in Point class (x- & y-coordinates)
// converts the object into a printable string
public String toString() {
   return “(“ + this.x + “,” + this.y + “)”;
}
```

Getters and Setters

- **Accessor ("getter") returns** the instance variable's value (takes in no parameters).
  Naming convention: `get + name of variable`
  - Example: In a Point class, get the x-coordinate instance variable "x"

```
public double getX() {
   return this.x;
}
```

- **Mutator ("setter") changes** (or sets) the value of an instance variable (takes in one parameter – the new value of the instance variable). Void return type!
  Naming convention: `set + name of variable`
  - Example: in a Circle class, set the "radius" instance variable to a new value

```
public void setRadius( double newRadius ) {
   this.radius = newRadius;
}
```

Accessors and Mutators (a.k.a. Getters and Setters) Another example – Cat class

- **Getters** and **setters** provide ways to access and change class fields
  - Supporting encapsulation, hiding how it is stored

```
public class Cat {
   private String name;
   public String getName() {
      ...
   }
   public void setName(String name) {
      ...
   }
}
```

Naming convention:

* Getters:

- “**get**” followed by the field name (e.g. `getName()`)

[retrieve the value of the field]

* Setters:

- “**set**” followed by the field name (e.g. `setName()`)

[alter the value of the field]

Other examples of object-oriented design

- Using Arrays
  - `int[] arr = new int[5]; // new keyword`
  - `arr.length; // Get the length of the array instance “arr”`

- Using Strings

  ```
  • s.equals(t);         // Using String s, see if it equals String s
  • s.toUppercase();     // return what the String s would look like if all
                         // the lowercase letters were uppercase
  • s.charAt(5);         //get the character at index 5 of String s
  ```

- The dot operator just means
  “Using this **instance**, get this value, or perform this action.”

Instance vs. Class

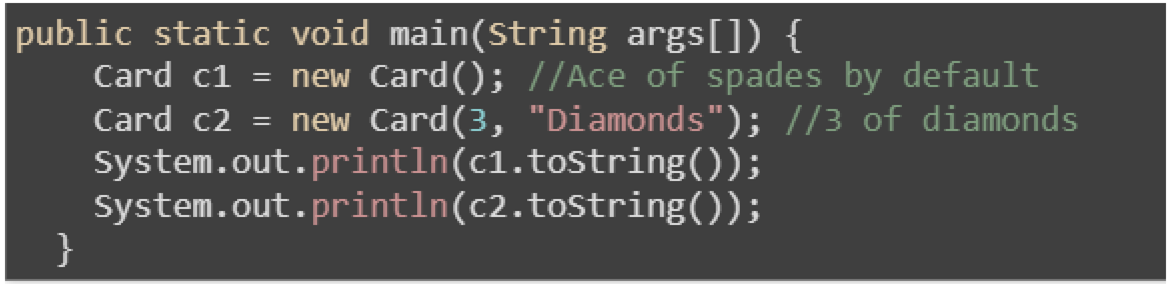
- Consider these lines of code:

```
String s = new String("Hi");  // while you don't need to use the new keyword for String
String t = new String("Bye"); // Java will use it implicitly (that is, hidden from view)
```

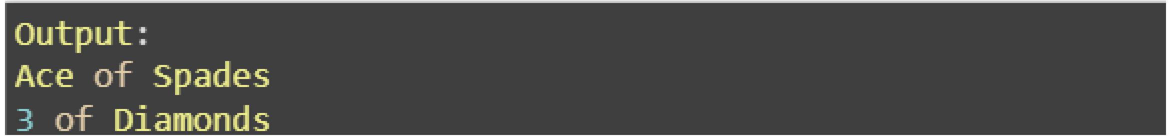
- **String** is the **class**
  - It contains all the code for **how a String works**
  - It lists all the fields that each String has
- ***s*** and ***t*** are **instances** of String
  - While ***s*** and ***t*** have the same behavior and the same set of variables
  - ***s*** and ***t*** have their own copy of the variables
    - That is, the “contents” of String ***s*** are separate from String ***t***
- All instances of a class share the same behaviors, but have their **own set** of the class’s variables

Back to Card Class…

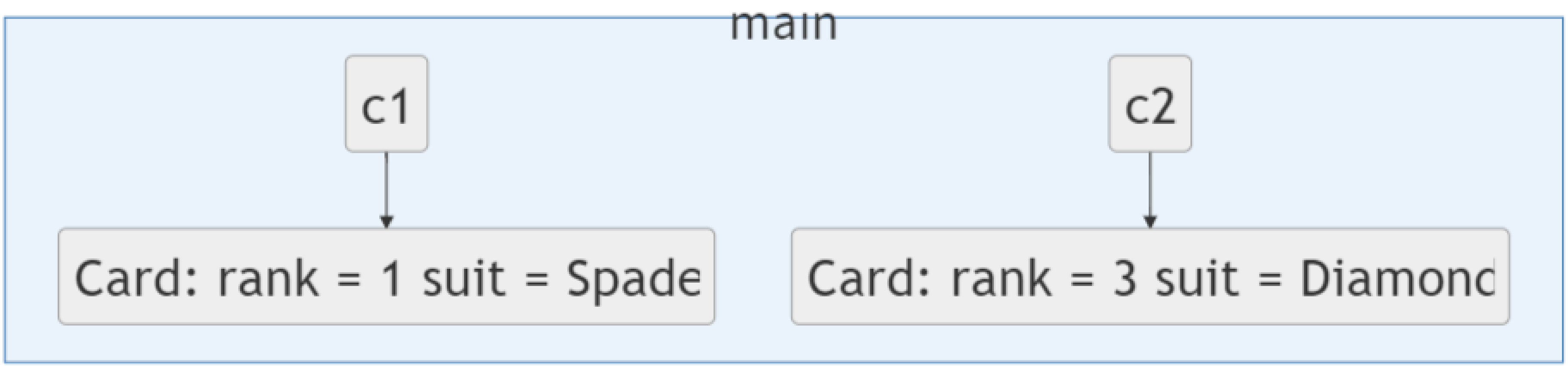
- Using the Card Class



```
public static void main(String args[]) {
    Card c1 = new Card(); //Ace of spades by default
    Card c2 = new Card(3, "Diamonds"); //3 of diamonds
    System.out.println(c1.toString());
    System.out.println(c2.toString());
  }
```

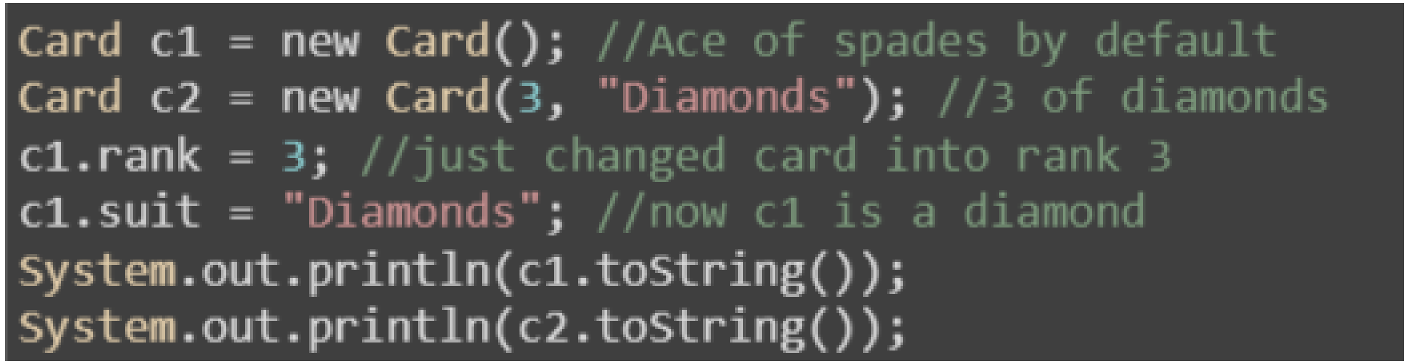


```
Output:
Ace of Spades
3 of Diamonds
```

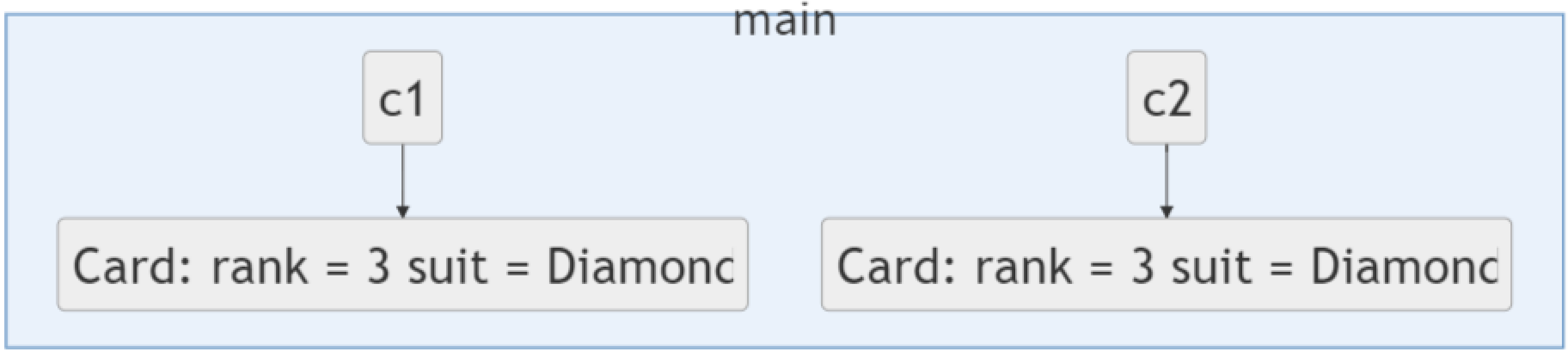


Accessing Fields

- You can access fields directly, for example changing Card c1's rank and suit:

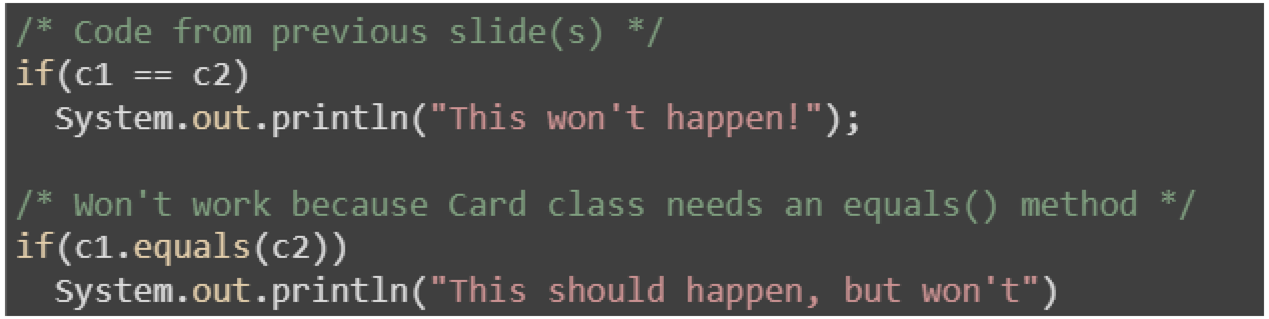


```
Card c1 = new Card(); //Ace of spades by default
Card c2 = new Card(3, "Diamonds"); //3 of diamonds
c1.rank = 3; //just changed card into rank 3
c1.suit = "Diamonds"; //now c1 is a diamond
System.out.println(c1.toString());
System.out.println(c2.toString());
```



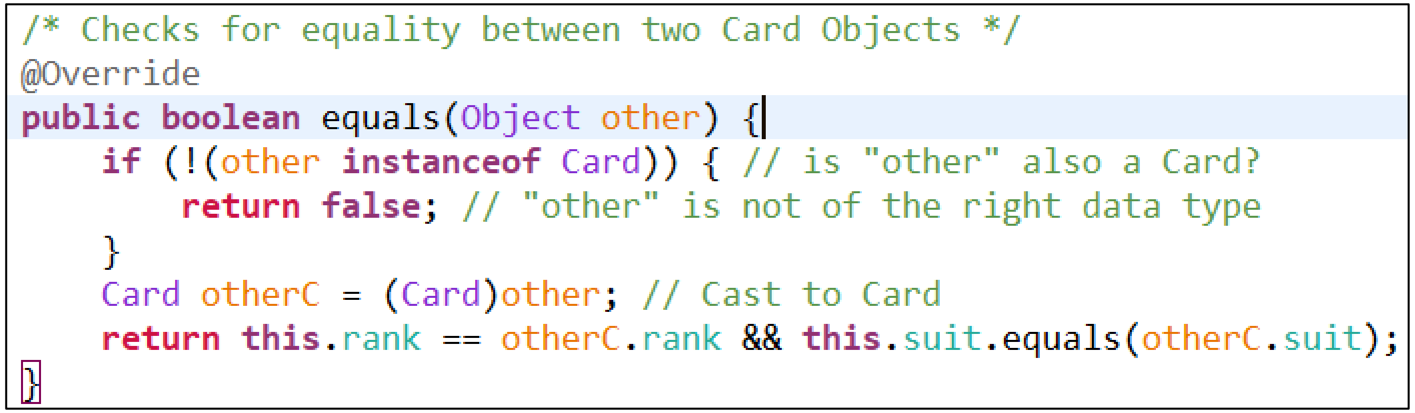
Checking Equality

- You cannot use the == operator to successfully compare two Objects (reference types)
- You MUST use the **.equals()** method instead
  - However…



```
/* Code from previous slide(s) */
if(c1 == c2)
  System.out.println("This won't happen!");

/* Won't work because Card class needs an equals() method */
if(c1.equals(c2))
  System.out.println("This should happen, but won't")
```

  - We must write our OWN equals() method in the Card Class:



```
/* Checks for equality between two Card Objects */
@Override
public boolean equals(Object other) {
    if (!(other instanceof Card)) { // is "other" also a Card?
        return false; // "other" is not of the right data type
    }
    Card otherC = (Card)other; // Cast to Card
    return this.rank == otherC.rank && this.suit.equals(otherC.suit);
}
```

Summary So Far

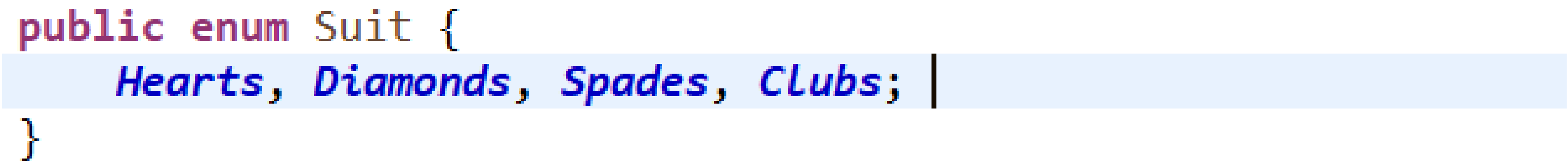
- **Card Class** looks pretty good so far
  - Has a rank and suit, can check equality, and can print

- **Next improvements:**
  - The suit can be ANY String (e.g., “BLAHBLAH”)
  - Rank can be any integer (e.g., -168)

- We can prevent our class variables from being **assigned incorrect values** in a couple of ways
  - **Enums**: Useful when a variable has a small, finite number of possible values (e.g., suit)

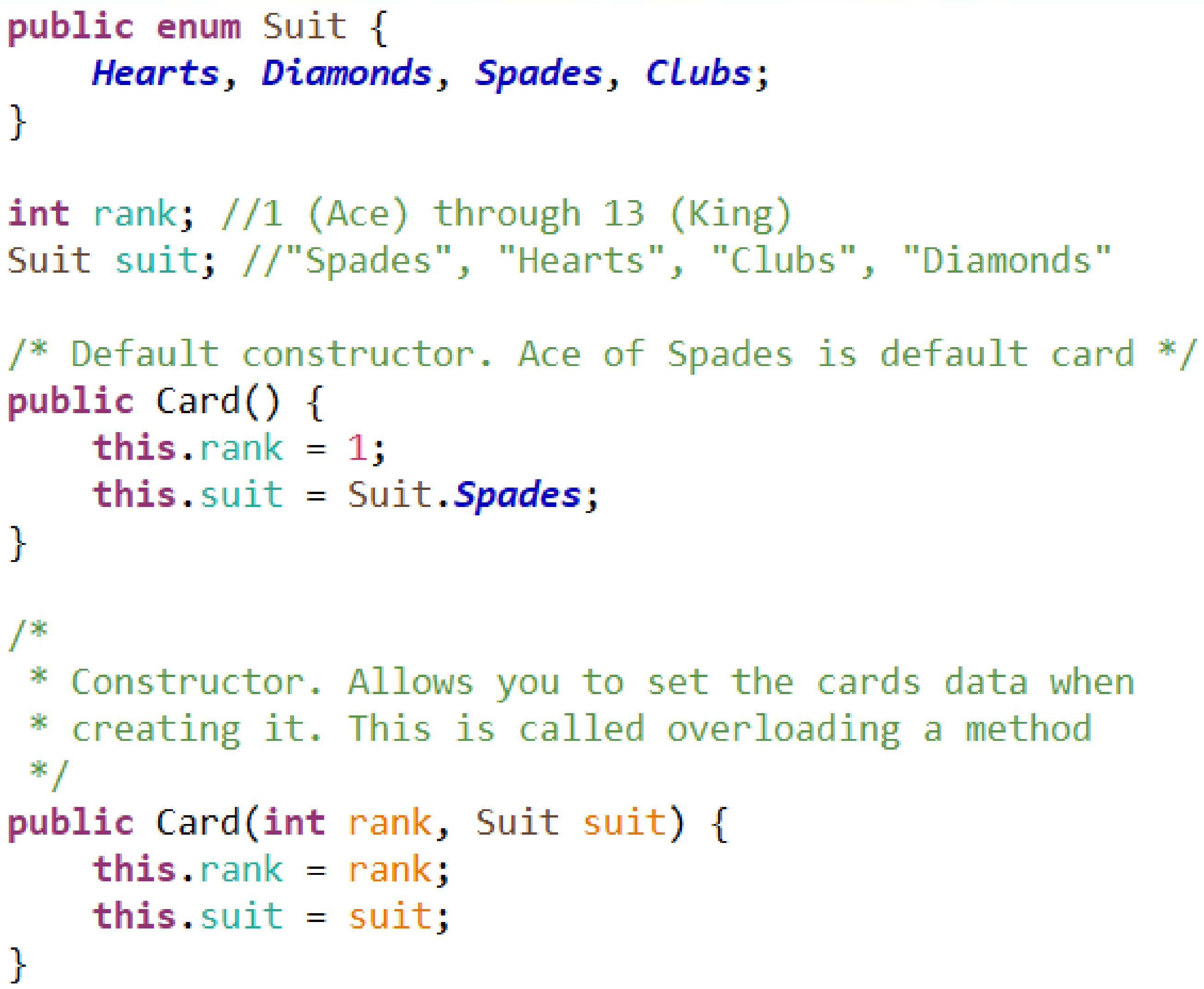
Enums

- **Enum** is short for "***enumerations***", which means "specifically listed".
- An **enum** is a special "class" that represents a group of **constants** (unchangeable variables)
- We can use an **enum** for the **suit of a Card:**



```
public enum Suit {
    Hearts, Diamonds, Spades, Clubs;
}
```

Changes to Card Class after using the enum

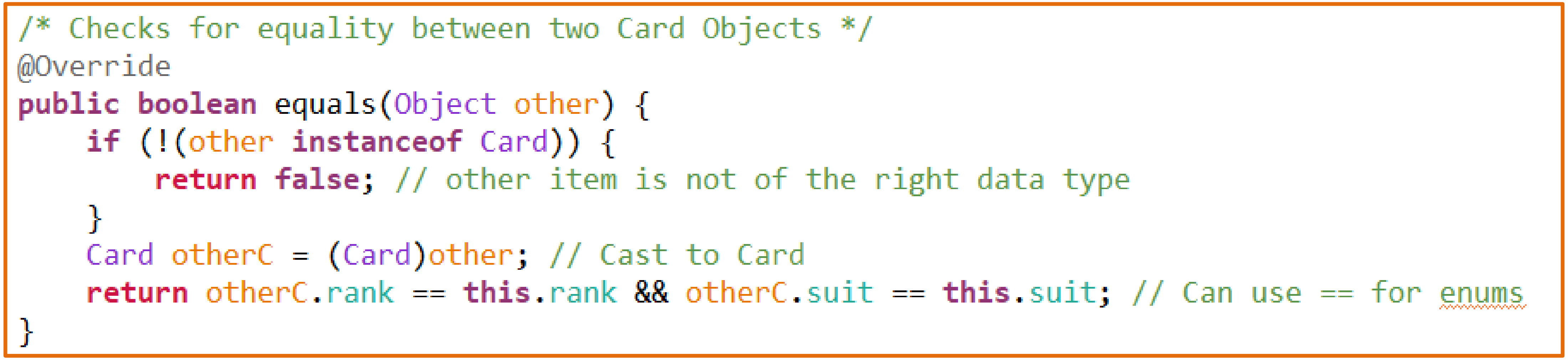


```
public enum Suit {
    Hearts, Diamonds, Spades, Clubs;
}

int rank; //1 (Ace) through 13 (King)
Suit suit; //"Spades", "Hearts", "Clubs", "Diamonds"

/* Default constructor. Ace of Spades is default card */
public Card() {
    this.rank = 1;
    this.suit = Suit.Spades;
}

/*
 * Constructor. Allows you to set the cards data when
 * creating it. This is called overloading a method
 */
public Card(int rank, Suit suit) {
    this.rank = rank;
    this.suit = suit;
}
```

Equality of Enums

- You can technically use == or .equals() though == is null safe and often preferred. WHY?
  - Because there is only one instance of each enum constant, it is permissible to use the == operator in place of the equals method when comparing two object references if it is known that at least one of them refers to an enum constant. (The equals method in Enum is a final method that merely invokes super.equals on its argument and returns the result, thus performing an identity comparison (calling .equals in Object class).)

Reference: https://docs.oracle.com/javase/specs/jls/se9/html/jls-8.html#jls-8.9



```
/* Checks for equality between two Card Objects */
@Override
public boolean equals(Object other) {
    if (!(other instanceof Card)) {
        return false; // other item is not of the right data type
    }
    Card otherC = (Card)other; // Cast to Card
    return otherC.rank == this.rank && otherC.suit == this.suit; // Can use == for enums
}
```

Summary So Far (continued)

- **Enum** is just a variable type that can take on a specified set of values. Otherwise, acts like any other variable.
- **Now suit can only be one of the four proper suits**.
- What about rank?
- We want it to be an **int** so we can compare it easily with other card ranks (can do this with enums, but a bit of a pain…)
- Another option is to **use getters / setters**, so let's see an example

Protecting our Fields

- Problem:
  - We don't want other programmers messing with the fields of our Card class and changing the values to unexpected things or illegal things (e.g., rank = -168)
- Solution:
  - Make the fields have **private** scope, then no one can mess with them.
  - Fields are set initially in the **constructor** only, **can't** be accessed afterwards
  - Provide **methods** to **access** and/or **set** the fields if necessary.

Access Specifiers / Visibility Modifiers

- Both **methods** and **attributes/variables** have access specifiers / visibility modifiers (sometimes discussed in the context of "scope")

modifier	class	package	sub-class	world
PUBLIC	Yes	Yes	Yes	Yes
PROTECTED	Yes	Yes	Yes	No
PRIVATE	Yes	No	No	No

- **public**: Anybody can access the field/method. Implied public if not specified
- **private**: Can only access from within the class definition
- **protected**: Can access from within same package or inheritance line
  - We won't use this for a little while

Updating Card Class

```
public class Card {

  /* An Enum is a variable type that has a finite set of values
   * Let’s use one for the suit of a Card
   */
  public enum Suit {
        Hearts, Diamonds, Spades, Clubs;
  }

  // Make class variables/fields *private*
  private int rank; //1 (Ace) through 13 (King)
  private Suit suit; //"Spades", "Hearts", "Clubs", "Diamonds"

  /* Default constructor. Ace of Spades is default card */
  public Card() {
    this.rank = 1;
    this.suit = Suit.Spades;
  }
```

Updating Card Class

```
/*
 * Constructor. Allows you to set the cards data when
 * creating it. This is called overloading a method
 */
public Card(int rank, Suit suit) {
  this.setRank(rank);
  setSuit(suit);
}
// GETTERS AND SETTERS:
public int getRank() { return this.rank; }
public Suit getSuit() { return this.suit; }

private void setRank(int newRank) {
  /* Ignore if trying to set to illegal value */
  if(newRank < 1 || newRank > 13) return;

  /* Otherwise, set it */
  this.rank = newRank;
}
private void setSuit(Suit newSuit) {
  /* No stress, enum already must have valid value */
  this.suit = newSuit;
}
/* Other stuff here... */
```

Summary

- Getters and Setters let us have control over how much other programmers can alter the fields in our class
- Maybe we don't want to be able to change a Card's suit and rank once it is instantiated
    - Solution: **Remove** the setters from the class
    - Now programmers can **see** the variables (through the **getters**) but not change them.

Deck Class

We’ll check out a more advanced Class next!

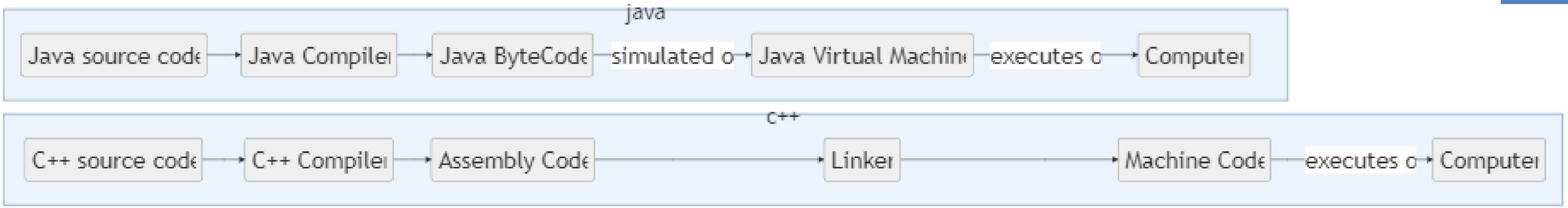
(Code posted along with Card Class)

JVM and Java API

JVM=Java Virtual Machine

Java vs C++

- The **Java Compiler** converts **Java source code** into **Java ByteCode**.
- **Java ByteCode** is simulated on the **Java Virtual Machine**, which executes on the **Computer**



Using the Java API

- Documentation of Java classes, methods, etc.
  - VERY useful for discovering what functionality already exists in Java and how to use it.

- Some examples:
  - Object: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html
  - Scanner: https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html
  - String: https://docs.oracle.com/javase/8/docs/api/java/lang/String.html
  - ArrayList: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html