Classes (1)

Han-fen Hu

Outline

- Writing a Class based on Class Diagram
- Constructors
- Overloading Methods and Constructors

Review: Model Class

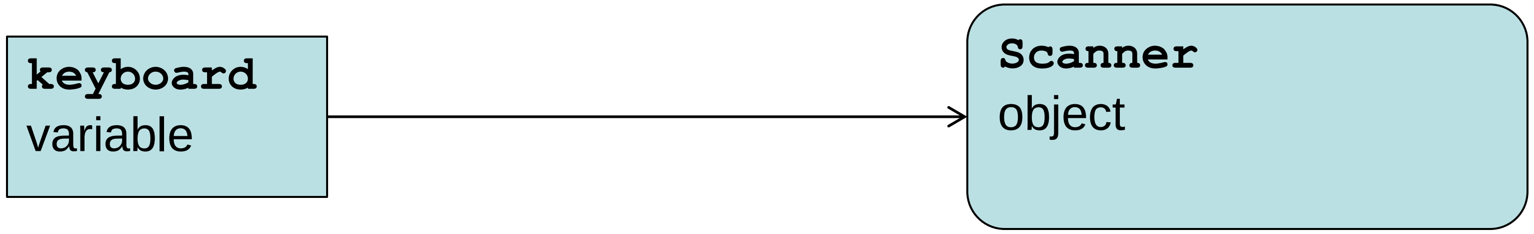
- Model class is a class that serves as a template to instantiate objects
  - Describes a particular type of object.
  - Specifies the data that an object can hold (the object's fields), and the actions that an object can perform (the object's methods).
- You have already used objects:
  - `Scanner` objects, for reading input
  - `PrintWriter` objects, for writing data to files

Instantiate Objects

This expression creates a **`Scanner`** object in memory.

```
Scanner keyboard = new Scanner(System.in);
```

The object's memory address is assigned to the **`keyboard`** variable.



Why Model Class?

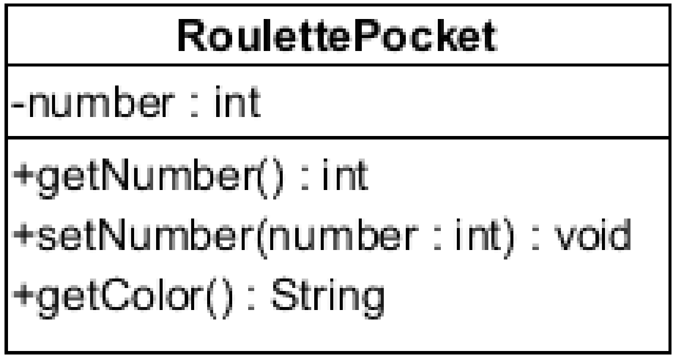
- Segregate model and controller
  - Model: representing the objects
  - Controller: Handling the flow of the application
- Model classes are established on the basis of data components and the relevant behavior
- The foundation of Model–View–Controller (MVC) design pattern

Example: Roulette Wheel

On a roulette wheel, the pockets are numbered from 0 to 36, The colors of the pockets are as follows:

- Pocket 0 is green.
- For pocket 1 through 10, the odd numbered pockets are red and the even numbered pockets are black.
- For pockets 11 through 18, the odd numbered pockets are black and the even numbered pockets are red.
- For pockets 19 through 28, the odd numbered pockets are red and the even numbered pockets are black.
- For pockets 29 through 36, the odd numbered pockets are black and the even numbered pockets are red.

Writing a Class (1)

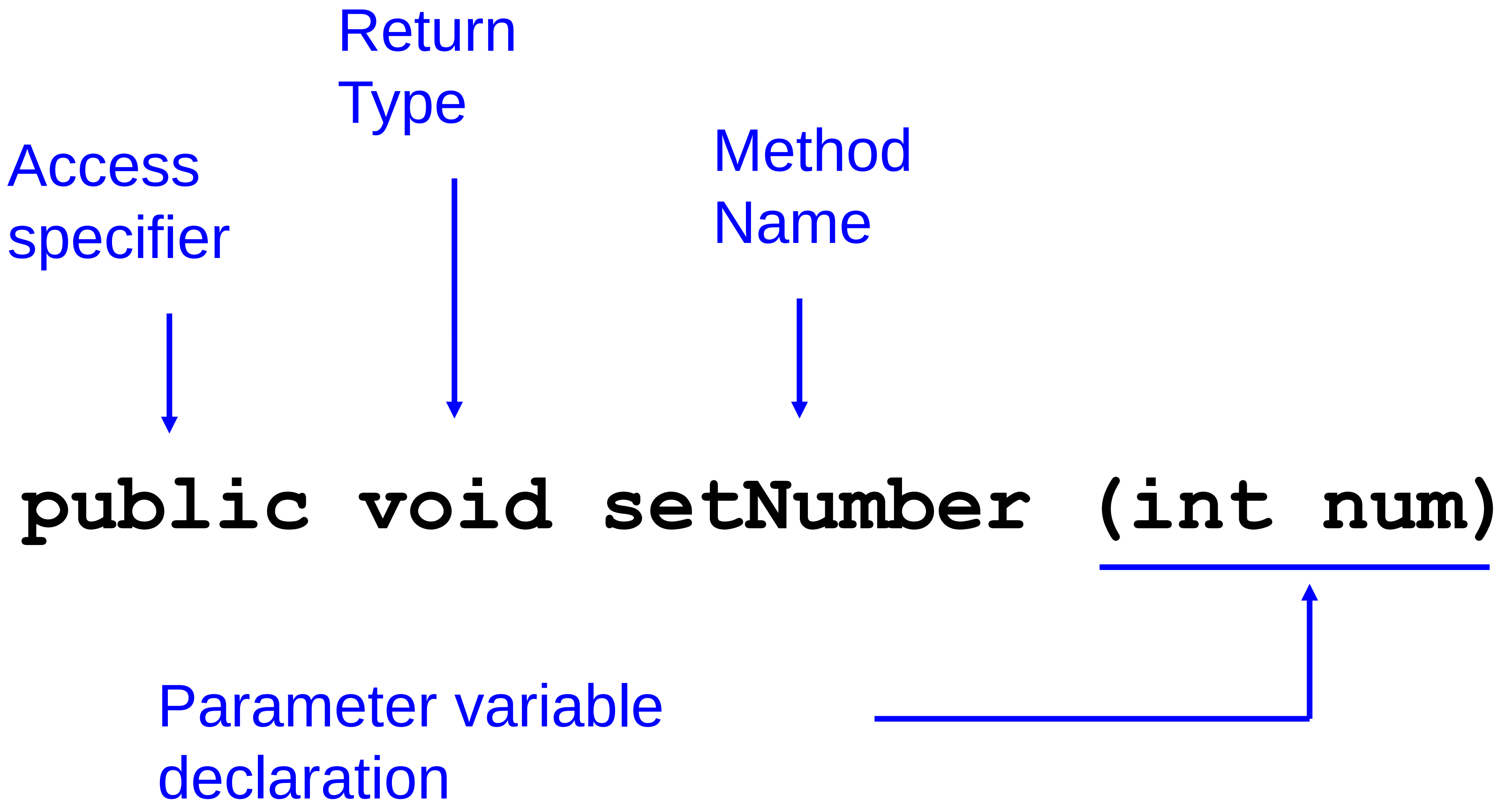


- A **RoulettePocket** object will have the following:
  - **`Number:`** holds the pocket's number
  - `getColor():` return the color, which is determined by the number

Access Specifiers

- An access specifier is a Java keyword that indicates how a field or method can be accessed.
  - **`public`**
    - When the **`public`** access specifier is applied to a class member, the member can be accessed by code inside the class or outside.
  - **`private`**
    - When the **`private`** access specifier is applied to a class member, the member cannot be accessed by code outside the class.
    - The member can be accessed only by methods that are members of the same class.

Header for the **`setNumber`** Method

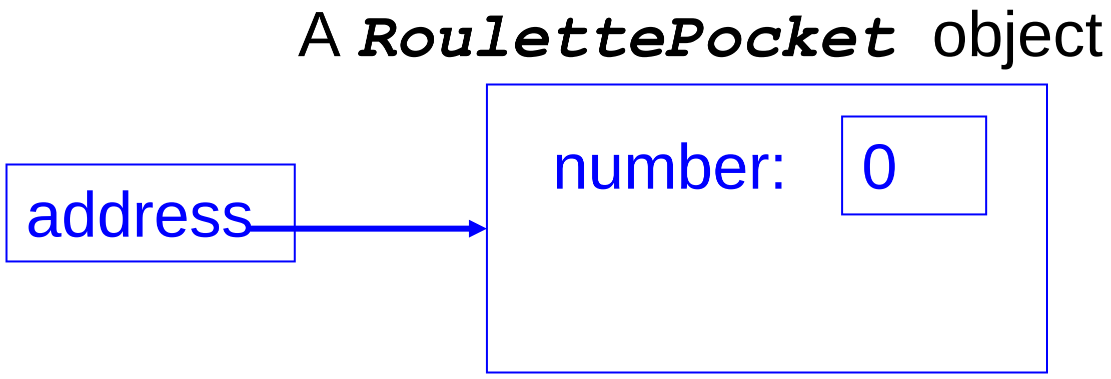


```
public void setNumber (int num)
```

Creating a **RoulettePocket** object

```
RoulettePocket pocket = new RoulettePocket();
```

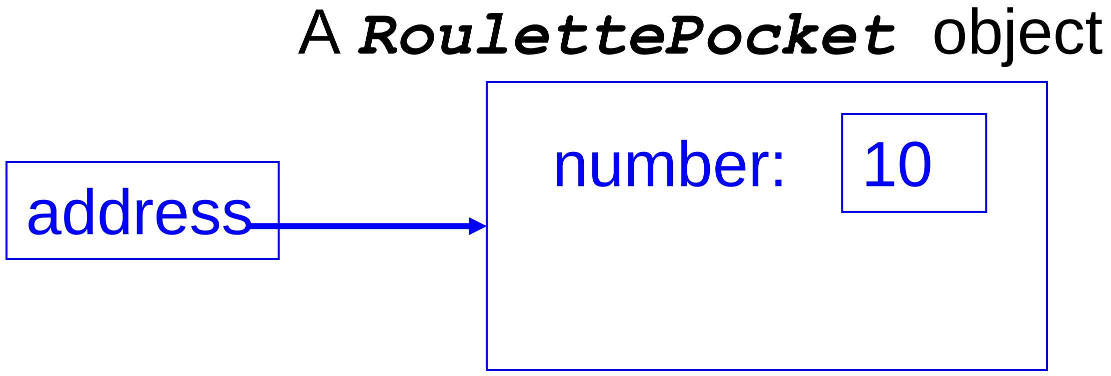
The **`pocket`** variable holds the address of the ***`RoulettePocket`*** object.



Calling the **`setNumber`** Method

`pocket.setNumber(10);`

The **`pocket`** variable holds the address of the ***`RoulettePocket`*** object.



This is the state of the **`pocket`** *object after the* **`setNumber`** *method executes.*

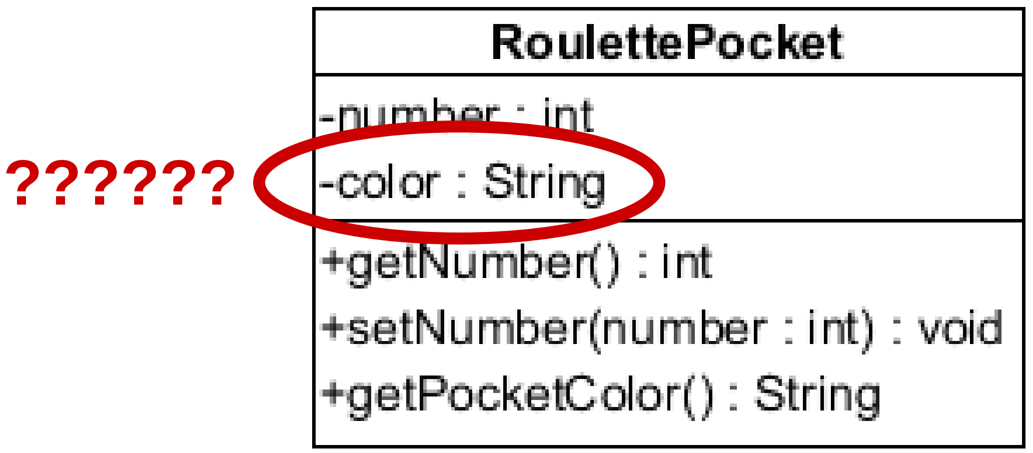
Review: Data Hiding (1)

- An object hides its internal, private fields from code that is outside the class that the object is an instance of.
- Only the class's methods may directly access and make changes to the object’s internal data.
- Code outside the class must use the class's public methods to operate on an object's private fields.

Review: Data Hiding (2)

- Data hiding is important because classes are typically used as components in large software systems, involving a team of programmers.
- Data hiding helps enforce the integrity of an object's internal data.

Discussion

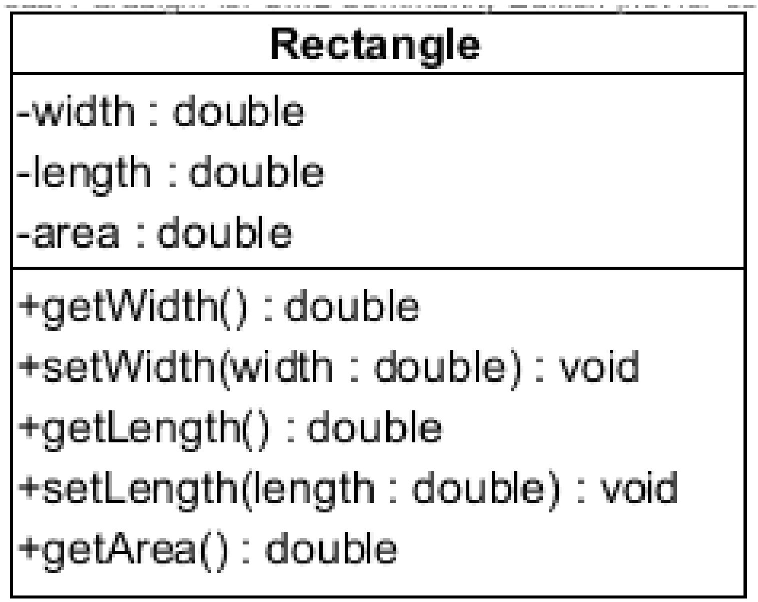


- Should there be a field “color”? Why or why not?

Stale Data

- Some data is the result of a calculation.
  - Consider the color of a RoulettePocket is determined by the number.
- Data that requires the calculation of various factors has the potential to become stale
- To avoid stale data, it is best to calculate the value of that data within a method rather than store it in a field
- getColor() method dynamically determine the value of the pocket’s color when the method is called

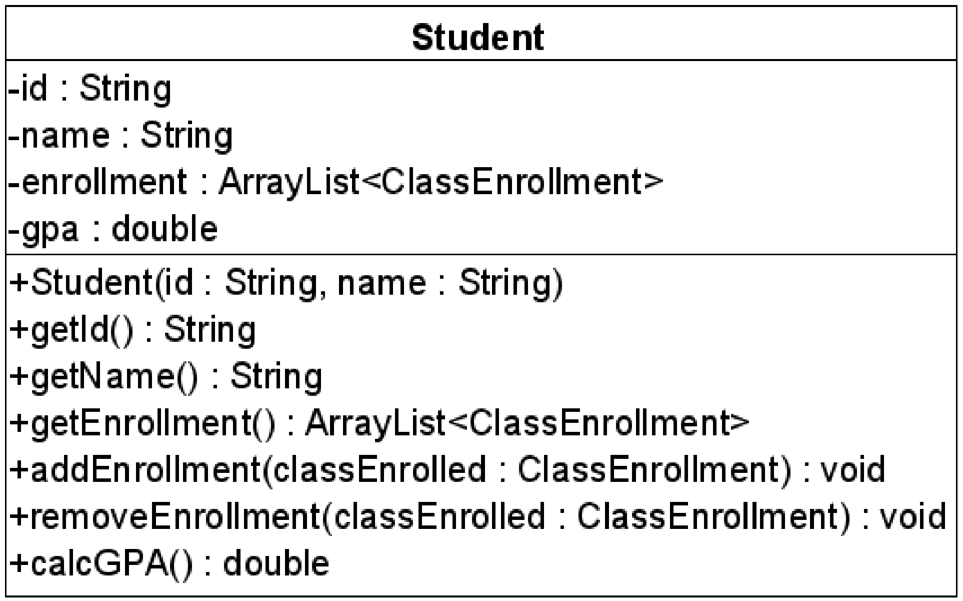
More Examples of Stale Data



Rectangle

-width : double
-length : double
-area : double

+getWidth() : double
+setWidth(width : double) : void
+getLength() : double
+setLength(length : double) : void
+getArea() : double



Student

-id : String
-name : String
-enrollment : ArrayList<ClassEnrollment>
-gpa : double

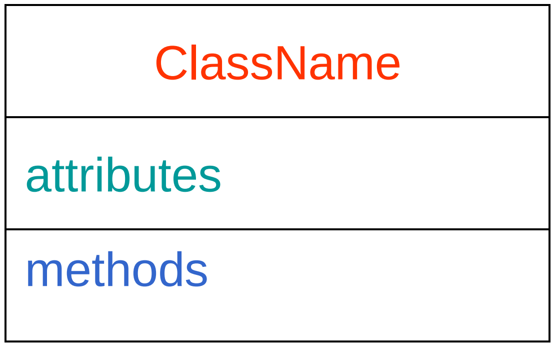
+Student(id : String, name : String)
+getId() : String
+getName() : String
+getEnrollment() : ArrayList<ClassEnrollment>
+addEnrollment(classEnrolled : ClassEnrollment) : void
+removeEnrollment(classEnrolled : ClassEnrollment) : void
+calcGPA() : double

- area of a rectangle should be calculated, rather than a static data field
- GPA should be calculated based on the class enrollment

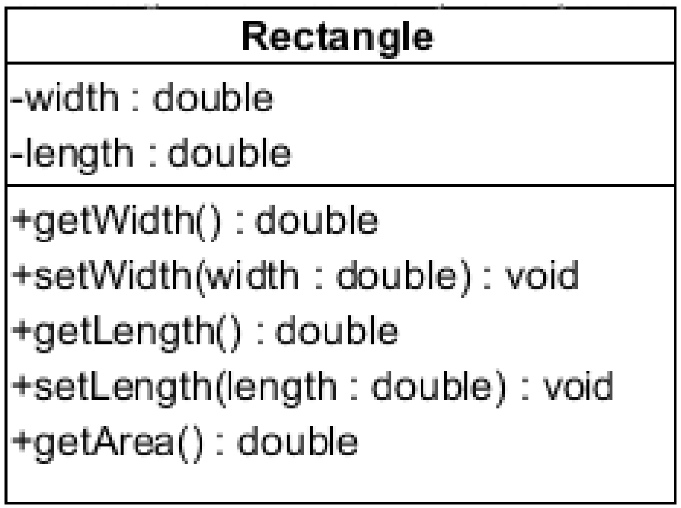
Converting Class Diagram to Code (1)

- Putting all of this information together, a Java class file can be built easily using the class diagram.
- The class diagram parts match the Java class file structure.

```
class header {
    fields
    methods
}
```



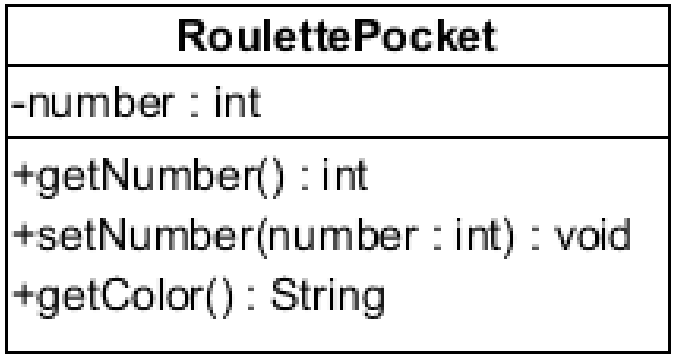
Converting Class Diagram to Code (2)

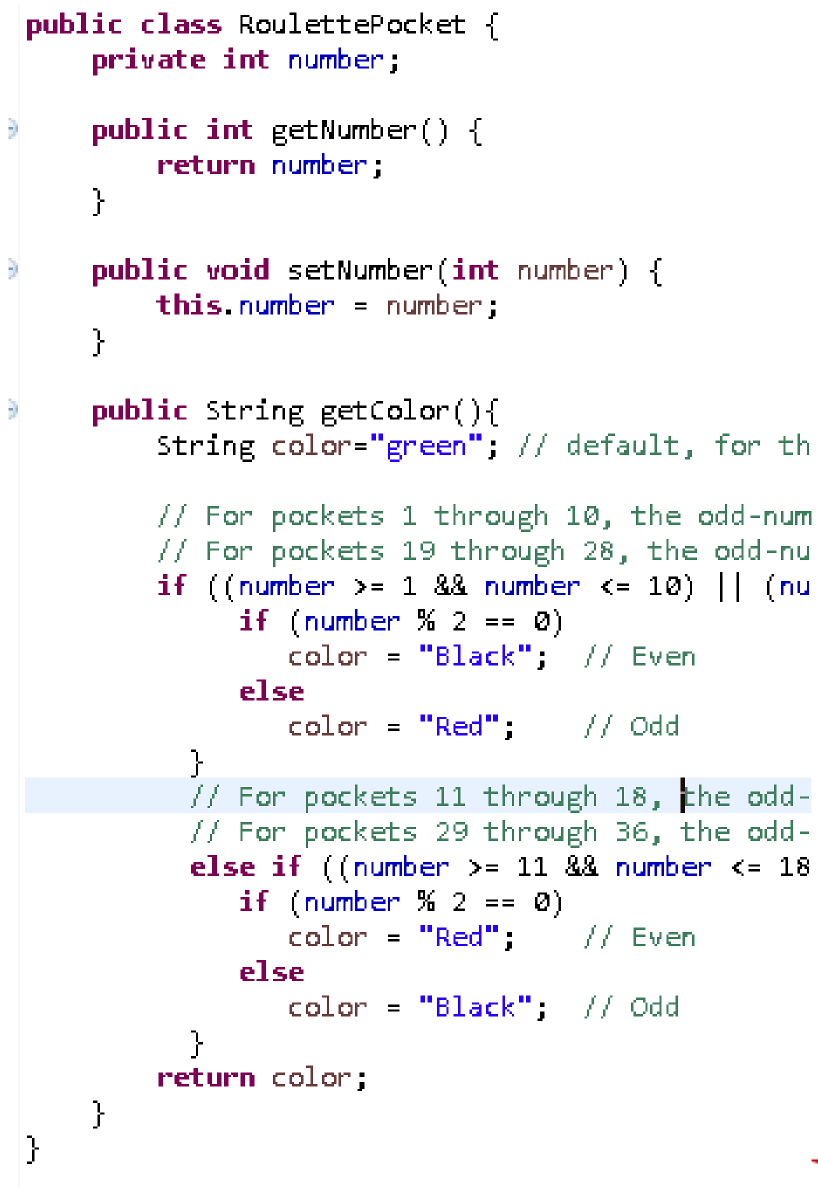


```
public class Rectangle {
    private double width;
    private double length;

    public double getWidth() {
        return width;
    }
    public void setWidth(double width) {
        this.width = width;
    }
    public double getLength() {
        return length;
    }
    public void setLength(double length) {
        this.length = length;
    }
    public double getArea() {
        return width * length;
    }
}
```

Converting Class Diagram to Code (3)




```
public class RoulettePocket {
    private int number;

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public String getColor(){
        String color="green"; // default, for th

        // For pockets 1 through 10, the odd-num
        // For pockets 19 through 28, the odd-nu
        if ((number >= 1 && number <= 10) || (nu
            if (number % 2 == 0)
                color = "Black";  // Even
            else
                color = "Red";    // Odd
          }
          // For pockets 11 through 18, the odd-
          // For pockets 29 through 36, the odd-
          else if ((number >= 11 && number <= 18
            if (number % 2 == 0)
                color = "Red";    // Even
            else
                color = "Black";  // Odd
          }
        return color;
    }
}
```

Class Layout Conventions

- A common layout is:
  - Fields listed first
  - Methods listed second
    - Set and Get methods are typically grouped.

Lab (1)

- RoulettePocket.java
- RoulettePocketDemo.java
  - Based on the class diagram on the previous slide, we will create the model class RoulettePocket in Java

Constructors (1)

- Classes can have special methods called constructors.
  - A constructor is a method that is automatically called when an object is created.
  - Constructors are used to perform operations at the time an object is created.
- Constructors typically initialize instance fields and perform other object initialization tasks.

Constructors (2)

- Constructors have a few special properties that set them apart from normal methods.
  - Constructors have the same name as the class.
  - Constructors have no return type (not even void).
  - Constructors may not return any values.
  - Constructors are typically public.

Constructor for **`RoulettePocket`** Class

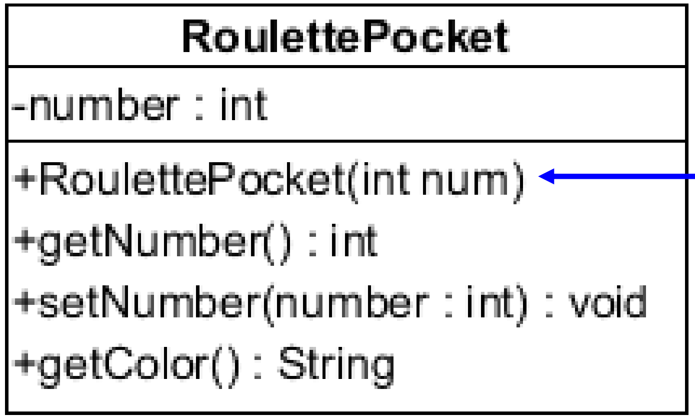
```
public RoulettePocket(int num){
      number = num;
}
```

Lab (2)

- RoulettePocket.java
  - We can now add constructors to the class. In the second constructor, we also added a decision structure to make sure the number will not be negative.

Constructors in Class Diagram

- In Class Diagram, the most common way constructors are defined is:



Notice there is no return type listed for constructors.

Default Constructor (1)

- When an object is created, its constructor is <u>always</u> called.
- If you do not write a constructor, Java provides one when the class is compiled. The constructor that Java provides is known as the *default constructor*.
  - It sets all of the object's numeric fields to 0.
  - It sets all of the object's **`boolean`** fields to **`false`**.
  - It sets all of the object's reference variables to the special value ***null***.

UNLV

Default Constructor (2)

- The default constructor is a constructor with no parameters, used to initialize an object in a default configuration.
- The only time that Java provides a default constructor is when you do not write any constructor for a class.
- A default constructor is not provided by Java if a constructor is already written.
  - Once you create a new constructor (with arguments), you need to create the default constructor on your own

Writing Your Own No-Arg Constructor

- A constructor that does not accept arguments is known as a *no-arg constructor*.
- The default constructor (provided by Java) is a no-arg constructor.
- We can write our own no-arg constructor

```
public RoulettePocket(){
     number = 0;
}
```

Or just leave the body blank.

```
public RoulettePocket(){
}
```

Lab (3)

- RoulettePocket.java
- RoulettePocketDemo.java
  - We already created the no-arg contructor in the last step. You can review the code now.

Instructions in No-Arg Constructor

- You can also implement the No-Arg constructor with instructions; it does not have to be an employ constructor
- For example, you can assign a random value or a default value when an object is instantiated

Lab (4)

- RoulettePocket.java
- RoulettePocketDemo.java
  - We can revise the no-arg constructor to assign a random number when the number is not passed.

Overloading Methods and Constructors

- Two or more methods in a class may have the same name as long as their parameter lists are different.
- When this occurs, it is called method overloading. This also applies to constructors.
- Method overloading is important because sometimes you need several different ways to perform the same operation.

Overloaded Method: Example

```
public int add(int num1, int num2){
   int sum = num1 + num2;
   return sum;
}

public String add (String str1, String str2){
   String combined = str1 + str2;
   return combined;
}
```

Method Signature and Binding

- A method signature consists of the method’s name and the data types of the method’s parameters, in the order that they appear.

```
add(int, int)
add(String, String)
```

Signatures of the `add` methods

- The return type is <u>not</u> part of the signature.
- The compiler uses the method signature to determine which version of the overloaded method to bind the call to.
  - The process of matching a method call with the correct method is known as *binding*.

`RoulettePocket` Class Constructor Overload

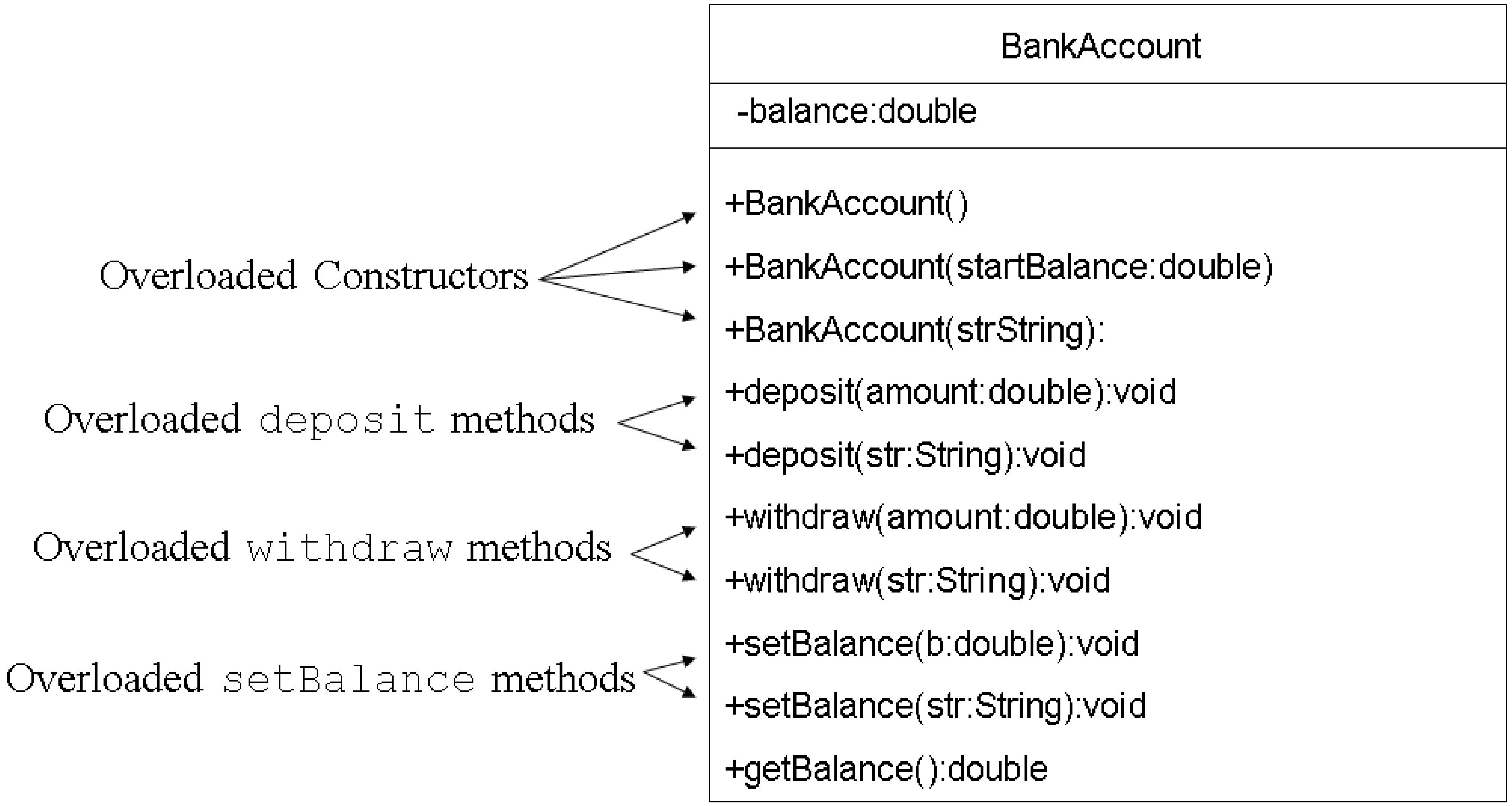
What would happen when we execute the following calls?

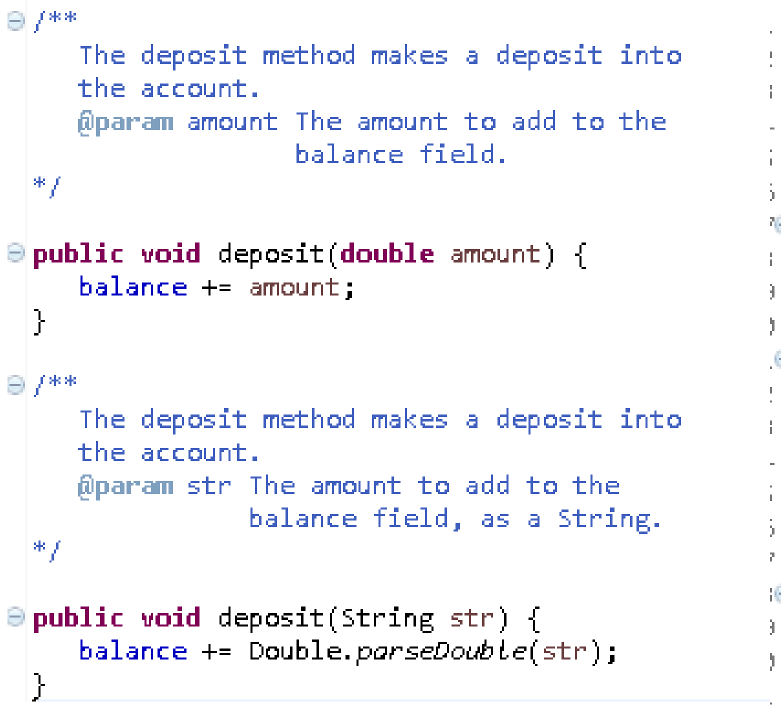
```
RoulettePocket pocket1 = new RoulettePocket();
RoulettePocket pocket2 = new RoulettePocket(3);
```

The first call would use the no-arg constructor and **`pocket1`** would have number set as 0

The second call would use the original constructor and **`pocket2`** would have the number set as 3

Overloaded Methods (1)


BankAccount
-balance:double
+BankAccount( )
+BankAccount(startBalance:double)
+BankAccount(strString):
+deposit(amount:double):void
+deposit(str:String):void
+withdraw(amount:double):void
+withdraw(str:String):void
+setBalance(b:double):void
+setBalance(str:String):void
+getBalance( ):double
Overloaded Constructors
Overloaded deposit methods
Overloaded withdraw methods
Overloaded setBalance methods

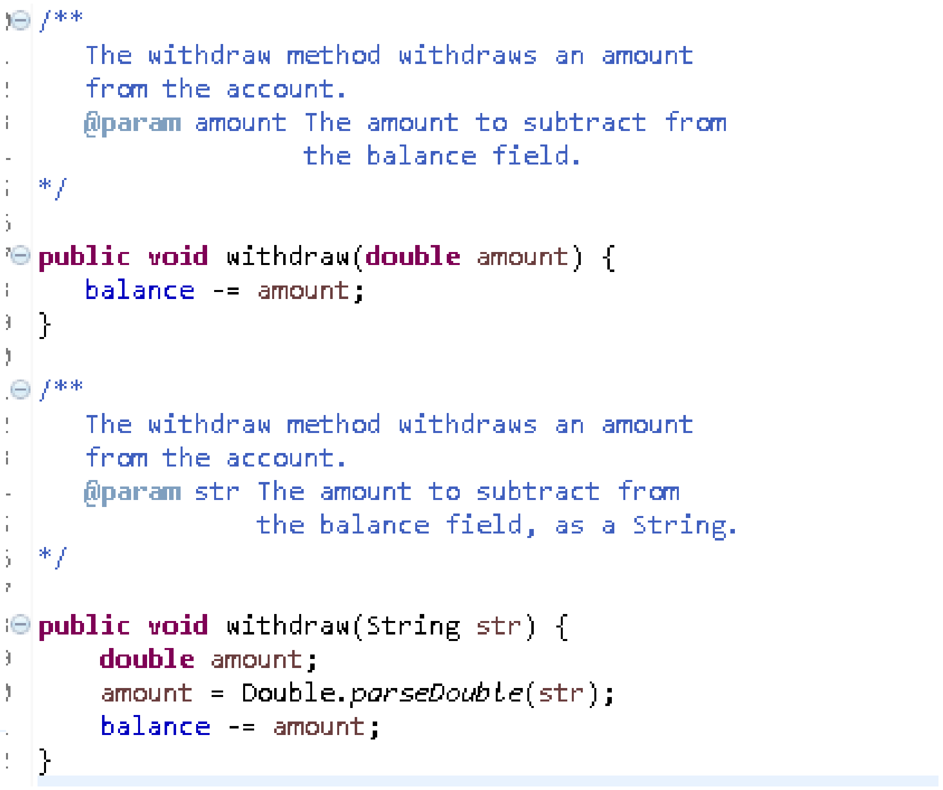


```
/**
    The deposit method makes a deposit into
    the account.
    @param amount The amount to add to the
                  balance field.
*/

public void deposit(double amount) {
    balance += amount;
}

/**
    The deposit method makes a deposit into
    the account.
    @param str The amount to add to the
               balance field, as a String.
*/

public void deposit(String str) {
    balance += Double.parseDouble(str);
}
```

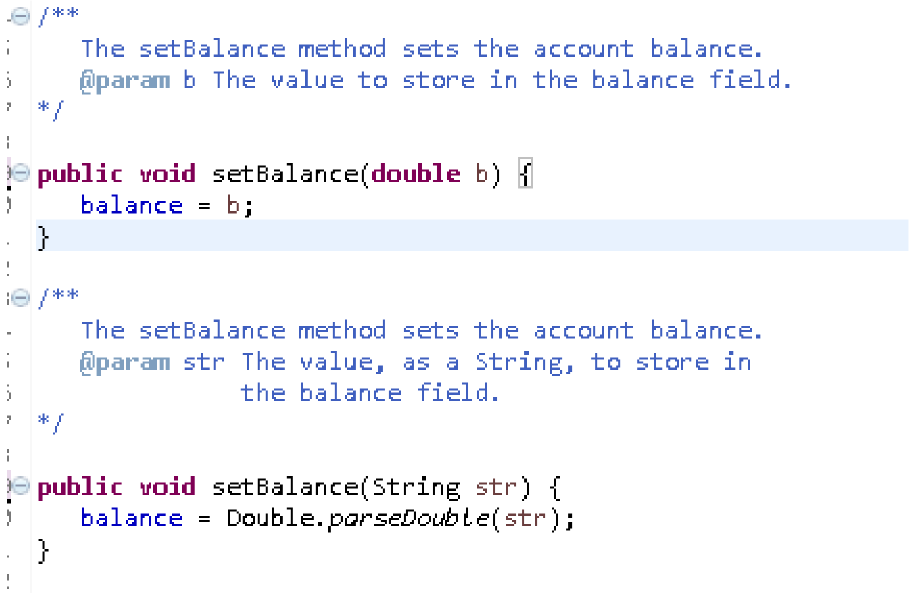


```
/**
    The withdraw method withdraws an amount
    from the account.
    @param amount The amount to subtract from
                  the balance field.
*/

public void withdraw(double amount) {
    balance -= amount;
}

/**
    The withdraw method withdraws an amount
    from the account.
    @param str The amount to subtract from
               the balance field, as a String.
*/

public void withdraw(String str) {
     double amount;
     amount = Double.parseDouble(str);
     balance -= amount;
}
```



```
/**
    The setBalance method sets the account balance.
    @param b The value to store in the balance field.
*/

public void setBalance(double b) {
    balance = b;
}

/**
    The setBalance method sets the account balance.
    @param str The value, as a String, to store in
               the balance field.
*/

public void setBalance(String str) {
    balance = Double.parseDouble(str);
}
```

Lab (5)

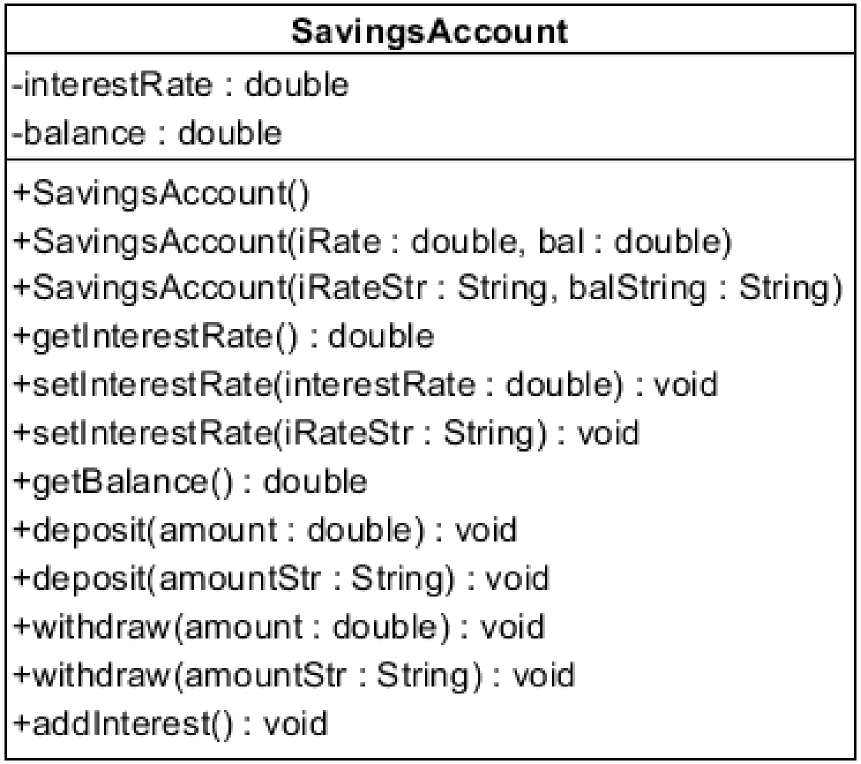
- RoulettePocket.java
  - Revise it to create an overloaded method setNumber() which accepts a String argument
- RoulettePocketDemo2.java
  - Complete the program to use a String in setting the number for the pocket object

Overloaded Methods (2)

- Overloaded methods make classes more useful
- It is much more flexible than it would be if it provided only one way to perform every operation

Lab (6)

- SavingsAccount
  - Based on the class diagram, please implement the class



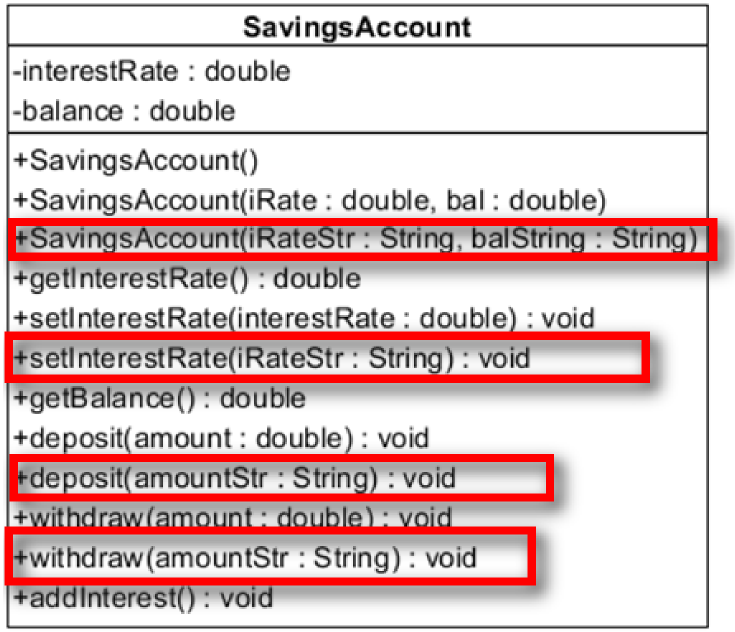
```
SavingsAccount
-interestRate : double
-balance : double
+SavingsAccount()
+SavingsAccount(iRate : double, bal : double)
+SavingsAccount(iRateStr : String, balString : String)
+getInterestRate() : double
+setInterestRate(interestRate : double) : void
+setInterestRate(iRateStr : String) : void
+getBalance() : double
+deposit(amount : double) : void
+deposit(amountStr : String) : void
+withdraw(amount : double) : void
+withdraw(amountStr : String) : void
+addInterest() : void
```

Detailed Logic of SavingsAccount

- No-arg constructor: Set the interest rate and balance to 0.
- Constructor: Accepts the interest rate and the amount of starting balance.
  - If starting balance is less than zero, set the field balance to zero (0).
  - If interest rate is less than 0, set the field interestRate to zero (0).
  - If interest rate is greater than 0.01 (i.e., 1%), divide the number by 100 before setting the value to the field.
- Get and set method for the interest rate, including the overloaded method with Strings
  - If the input interest rate is less than 0, set the field interestRate to zero (0).
  - If the input interest rate is greater than 0.01 (i.e., 1%), divide the number by 100 before setting the value.
- deposit() method: Add the amount of a deposit to the balance, including the overloaded method with String as the parameter. If the input amount is less than 0, set the amount to zero(0).
- withdraw() method: Subtract the amount of a withdrawal from the balance, including the overloaded method with String as the parameter. If the input amount is less than 0, set the amount to zero(0).
- addInterest() method: Adding the amount of monthly interest to the balance. To add the monthly interest to the balance, multiply the monthly interest rate by the balance, and add the result to the balance.

Class Exercise

- Please implement the other methods that accepts string arguments.
- Once you complete SavingsAcocunt class, you should be able to run SavingsAccountDemo to test the program



```
SavingsAccount
-interestRate : double
-balance : double
+SavingsAccount()
+SavingsAccount(iRate : double, bal : double)
+SavingsAccount(iRateStr : String, balString : String)
+getInterestRate() : double
+setInterestRate(interestRate : double) : void
+setInterestRate(iRateStr : String) : void
+getBalance() : double
+deposit(amount : double) : void
+deposit(amountStr : String) : void
+withdraw(amount : double) : void
+withdraw(amountStr : String) : void
+addInterest() : void
```