GET SMART: WITH JAVA PROGRAMMING

YAMAN OMAR ALASHQAR

SYSTEM.OUT.PRINTLN(“WELCOME TO THIS COURSE”);

OOP

CONTINUE

- Classes define what an object knows and what an object does.
- Things an object knows are its **instance variables** (state).
- Things an object does are its **methods** (behavior).
- Methods can use instance variables so that objects of the same type can behave differently.
- A method can have parameters, which means you can pass one or more values in to the method.
- The number and type of values you pass in must match the order and type of the parameters declared by the method.
- The value you pass as an argument to a method can be a literal value (2, ‘c’, etc.) or a variable of the declared parameter type (for example, *x* where *x* is an int variable). (There are other things you can pass as arguments, but we’re not there yet.)
- A method *must* declare a return type. A void return type means the method doesn’t return anything.
- If a method declares a non-void return type, it *must* return a value compatible with the declared return type

CONSTRUCTORS

A constructor in Java is a special method that is used to initialize objects.
The constructor is called when an object of a class is created.
It can be used to set initial values for object attributes

Note that the constructor name must **match the class name**,
and it cannot have a **return type**

Also note that the constructor is called when the object is created.

All classes have constructors by default:
if you do not create a class constructor yourself, Java creates one for you.

Constructors can also take parameters, which is used to initialize attributes

Method Overloading

Method Overloading

- Method overloading is a concept in which we can create multiple methods of the same name in the same class, and all methods work in different ways.
  - More than one method with the same name, but with different parameters.
  - Different Implementations

- In java, we do method overloading in two ways: –
  - By changing the number of parameters.
  - By changing data types.

Benefits of Method Overloading

- Method overloading increases the readability of the program.
- This provides flexibility to programmers so that they can call the same method for different types of data.
- This makes the code look clean.
- This reduces the execution time because the binding is done in compilation time itself.
- Method overloading minimizes the complexity of the code.
- With this, we can use the code again, which saves memory.

QUESTION

```
public class Circle {
 double radius;
 String color;
 public double getArea(){ return 0; }
 public static int test(){ return 10; }
}
```

- Recall that you use Math.methodName(arguments) (e.g., Math.pow(3, 2.5)) to invoke a method in the Math class. Can you invoke getArea() using Circle.getArea()?

The answer is no. All the methods in the Math class are static methods, which are defined using the static keyword.
However, getArea() is an instance method, and thus nonstatic. It must be invoked from an object using objectRefVar. methodName(arguments)
(e.g. myCircle.getArea()).

- The **static members** are **used** to store data independent of any instance of an object

One rule-of-thumb: ask yourself "Does it make sense to call this method, even if no object has been constructed yet?" If so, it should definitely be static.

So in a class `Car` you might have a method:

```
double convertMpgToKpl(double mpg)
```

...which would be static, because one might want to know what 35mpg converts to, even if nobody has ever built a `Car`. But this method (which sets the efficiency of one particular `Car`):

```
void setMileage(double mpg)
```

...can't be static since it's inconceivable to call the method before any `Car` has been constructed.

- The **static members** are **used** to store data independent of any instance of an object

If you are writing utility classes and they are not supposed to be changed.
If the method is not using any instance variable.

If any operation is not dependent on instance creation.

If there is some code that can easily be shared by all the instance methods, extract that code into a static method.

If you are sure that the definition of the method will never be changed or overridden. As static methods can not be overridden.

A static variable is shared by all objects of the class.
A static method cannot access instance members (i.e., instance data fields and methods) of the class

DIFFERENCES BETWEEN VARIABLES OF PRIMITIVE TYPES AND REFERENCE TYPES

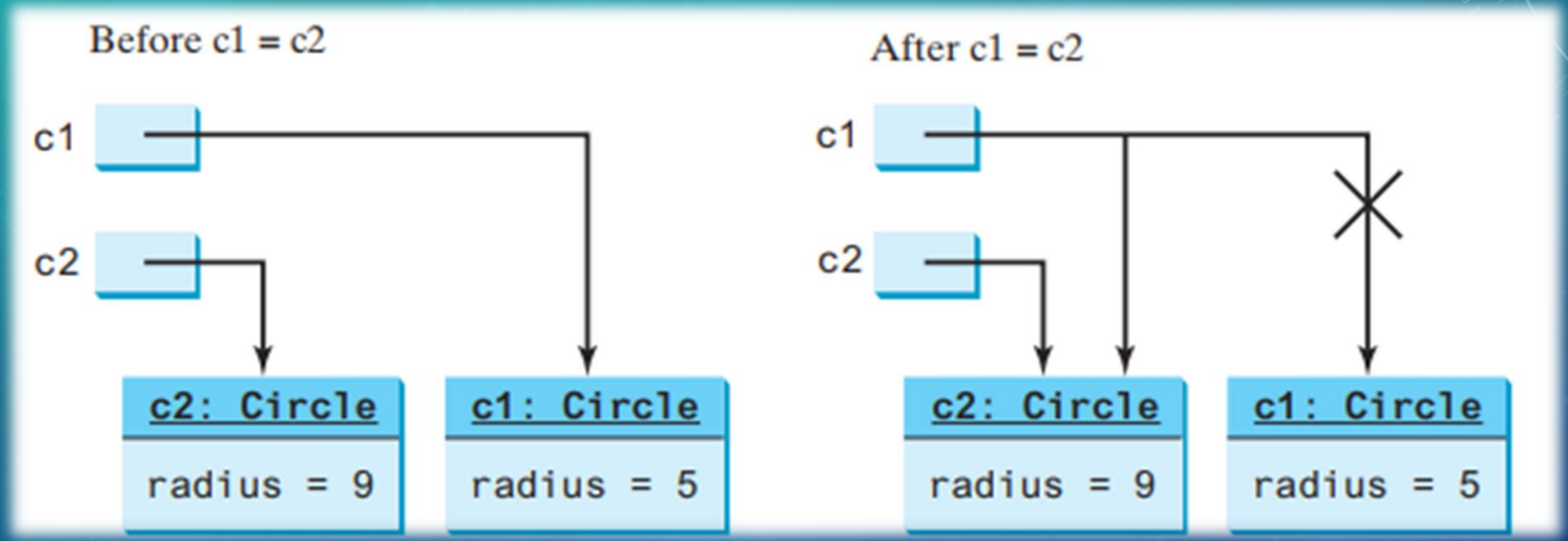
- Every variable represents a memory location that holds a value. When you declare a variable, you are telling the compiler what type of value the variable can hold.
- For a variable of a primitive type, the value is of the primitive type.
- For a variable of a reference type, the value is a reference to where an object is located.

DIFFERENCES BETWEEN VARIABLES OF PRIMITIVE TYPES AND REFERENCE TYPES

- When you assign one variable to another, the other variable is set to the same value.
- For a variable of a primitive type, the real value of one variable is assigned to the other variable.
- For a variable of a reference type, the reference of one variable is assigned to the other variable.

DIFFERENCES BETWEEN VARIABLES OF PRIMITIVE TYPES AND REFERENCE TYPES

- The assignment statement **i = j** copies the contents of **j** into **i**
- For primitive variables, the assignment statement c1 = c2 copies the reference of c2 into c1 for reference variables. After the assignment, variables c1 and c2 refer to the same object.


Before c1 = c2
c1
c2
c2: Circle
radius = 9
c1: Circle
radius = 5
After c1 = c2
c1
c2
c2: Circle
radius = 9
c1: Circle
radius = 5

CLASS BOOK

- Title
- Author
- Number of pages
- Description
- RATING
- PricePerDay
- Date of rent / Date of return
- calculatePrice(number of days)
- printInfo()
- PreviewRandomPage()

ACCESS MODIFIERS IN JAVA

- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it

ACCESS MODIFIERS IN JAVA

- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Access Modifier	Within Class	Within Package	Same Package by subclasses	Outside Package by subclasses	Global
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

DATA FIELD ENCAPSULATION

You are not allowed to access the totalAmount

DATA FIELD ENCAPSULATION

You are not allowed to access the numberOfCans, totalEarnings,
Also you are not allowed to set a new price for an item

DATA FIELD ENCAPSULATION

Making data fields private protects data and makes the class easy to maintain

- The data fields in a can be modified directly
- This is not a good practice—for two reasons
  - Data may be tampered with. (Imagine accessing a variable that represents a counter)
  - The class becomes difficult to maintain and vulnerable to bugs.

DATA FIELD ENCAPSULATION

Making data fields private protects data and makes the class easy to maintain

Suppose that you want to modify the Circle class to ensure that the radius is nonnegative after other programs have already used the class.

You have to change not only the Circle class but also the programs that use it because the clients may have modified the radius directly (e.g., c1.radius = –5).

DATA FIELD ENCAPSULATION

Making data fields private protects data and makes the class easy to maintain

To prevent direct modifications of data fields, you should declare the data fields private, using the **private** modifier. This is known as *data field encapsulation*.

A private data field cannot be accessed by an object from outside the class that defines the private field.

However, a client often needs to retrieve and modify a data field.
HOW?

DATA FIELD ENCAPSULATION

Making data fields private protects data and makes the class easy to maintain

To make a private data field accessible, provide a *getter* method to return its value.

To enable a private data field to be updated, provide a *setter* method to set a new value.

A getter method is also referred to as an accessor and a setter to a mutator

SETTER & GETTERS

- READ ONLY / WRITE ONLY
- STORE CERTAIN INFORMATION
    - *SUCH AS WHO/WHAT CHANGED THE 'BALANCE'*
- VALIDATE DATA
    - ADD RESTRECTIONS
        - *IF NOT 'LOGGED IN' -> CAN'T USE 'getInbox'*
        - *'Balance should not be changed anywhere'*
    - CHECK FOR CONDITIONS
        - *IF 'balance-ammount < 0' -> NOT ALLOWED*

EXAMPLE

- setTime(h,m,s)
- if(h>=0 && <= 12) && (m>0<60) && (s>0<60)
- else throw out of range

Example

- SETTER'S AND GETTER'S FOR 'BOOK' CLASS
  - ADD THE SUITABLE VALIDATION RULES
    - Title should be upper case (For the first letter)
    - CHALLENGE: EVERY WORD SHOULD START WITH A CAPITAL LETTER
    - ISBN VALIDATION
      - 13 IN LENGTH
      - OR 14 but with a '-' after 3 digits

A

```
class TapeDeck {

  boolean canRecord = false;

  void playTape() {
    System.out.println("tape playing");
  }

  void recordTape() {
    System.out.println("tape recording");
  }
}

class TapeDeckTestDrive {
  public static void main(String [] args) {

    t.canRecord = true;
    t.playTape();

    if (t.canRecord == true) {
      t.recordTape();
    }
  }
}
```

B

```
class DVDPlayer {

  boolean canRecord = false;

  void recordDVD() {
    System.out.println("DVD recording");
  }
}

class DVDPlayerTestDrive {
  public static void main(String [] args) {

    DVDPlayer d = new DVDPlayer();
    d.canRecord = true;
    d.playDVD();

    if (d.canRecord == true) {
      d.recordDVD();
    }
  }
}
```

PASSING OBJECTS TO METHODS

Passing an object to a method is to pass the reference of the object

- You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object

ARRAY OF OBJECTS

An array can hold objects as well as primitive-type values.

The Guessing Game

Summary:

The guessing game involves a 'game' object and three 'player' objects. The game generates a random number between 0 and 9, and the three player objects try to guess it. (We didn't say it was a really *exciting* game.)

Classes:

`GuessGame.class` `Player.class` `GameLauncher.class`

The Logic:

1) The GameLauncher class is where the application starts; it has the main( ) method.

2) In the main( ) method, a GuessGame object is created, and its startGame() method is called.

3) The GuessGame object's startGame() method is where the entire game plays out. It creates three players, then "thinks" of a random number (the target for the players to guess). It then asks each player to guess, checks the result, and either prints out information about the winning player(s) or asks them to guess again.