Where do objects come from?

Good question!

We will learn how to create a class that defines a new datatype, i.e., a new type of objects.

We need to learn:

1. What is the structure of a class definition?
2. How to specify what happens when an object is instantiated (i.e., when the `new` operator is used)?
3. How do we define the methods that can be invoked through objects of this class?

Example: Account datatype

- represents a generic bank account

- acct1
  - acctNumber: 72354
  - balance: 102.56
  - name: "Ted Murphy"

- acct2
  - acctNumber: 69713
  - balance: 40.00
  - name: "Jane Smith"
1. Structure of class definition

- **Account** class
- **Data**
  - `int acctNumber;
  - double balance;
  - String name;
- **Constructor**
  - deposit()
  - withdraw()
  - getBalance()
  - toString()

- The class is the **blueprint**
  - Classes define DATA and METHODS
  - i.e., a datatype

2. Object instantiation

Creating Objects – old example:
- We have already seen something like this:

```java
Scanner scan = new Scanner (System.in);
```

This invokes the **Scanner constructor**, which is a special method that sets up the object.

Creating Objects – our newly defined **Account** class:
- A new **Account** object is created!

```java
Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
```

Invokes the **Account constructor**, which is a special method that sets up the object.
3. Method invocation

- As we have seen, once an object has been created, we can use the **dot operator** to invoke its methods:

```java
ans = scan.nextLine();
numChars = title.length();
amount = acct1.getBalance();
acct1.deposit(25.85);
```

Writing Classes

- So far, we've been using predefined classes from the Java API. Now we will learn to write our own classes.
  - class definitions
  - constructors
  - instance data
  - method declaration and parameter passing
  - encapsulation and Java modifiers
  - more about creating graphical objects (next week)
```java
public void withdraw(double x, double fee) {
    balance = balance - x - fee;
}

double balance;
int acctNumber;
String name;
```
Designing Classes

Java Bootcamp - Villanova University

Transactions class: Creating Account objects

```java
public Account (String x, int y, double z) {
    name = x;
    acctNumber = y;
    balance = z;
}
```

Transactions class: Creating Account objects

```java
int acctNumber;
String name;
```

Transactions class: Creating more Account objects

```java
acct1.deposit (25.85);
```

Account class: Using methods

```java
acct1 = new Account ("Ted Murphy", 72354, 102.56);
acct2 = new Account ("Jane Smith", 69713, 40.00);
```
Account class: Using methods

```java
acct1.deposit (25.85);
//---------------------------------------------------
//  Deposits the specified amount into the account.
//---------------------------------------------------
public void deposit (double x) {
    balance = balance + x;
}
```

acctNumber 72354
balance 102.56
name "Ted Murphy"
Account class: Another Example

```java
cacct1.withdraw (60, 2.50);
```

```java
public void withdraw (double x, double fee) {
    balance = balance - x - fee;
}
```

Class definitions

- A class can contain data declarations and method declarations

```java
int acctNumber;
double balance;
String name;
```

Data declarations (also called *fields*)

Method declarations (note: the constructor is also a method)
Bank Account Example

- There are some improvements that can be made to the `Account` class
- The design of some methods could also be more robust, such as verifying that the `amount` parameter to the `withdraw()` method is positive
- Some of these improvements are in the book examples
  - `Account.java`, `Transactions.java` (simplified versions)
  - `Account.java`, `Transactions.java` (book versions)

Examples of datatypes (Classes)

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>Name, Address, Major, Grade, point average</td>
<td>Set address, Set major, Compute grade point average</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Length, Width, Color, Size</td>
<td>Set length, Set width, Set color</td>
</tr>
<tr>
<td>Aquarium</td>
<td>Material, Length, Width, Weight</td>
<td>Compute volume, Compute total weight</td>
</tr>
<tr>
<td>Flight</td>
<td>Airline, Right number, Destination city</td>
<td>Get airline, Get flight number, Determine status</td>
</tr>
<tr>
<td>Employee</td>
<td>Name, Department, Title, Salary</td>
<td>Set department, Set title, Compute wages, Compute bonus</td>
</tr>
</tbody>
</table>
Designing Classes

Dr. Papalaskari

Java Bootcamp - Villanova University

//********************************************************************
// RollingDice.java  Author: Lewis/Loftus
// Demonstrates the creation and use of a user-defined class.
//********************************************************************
public class RollingDice
{
    // Creates two Die objects and rolls them several times.
    public static void main (String[] args)
    {
        Die die1, die2;
        int sum;
        die1 = new Die();
        die2 = new Die();
        die1.roll();
        die2.roll();
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);
        die1.roll();
        die2.setFaceValue(4);
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);
        sum = die1.getFaceValue() + die2.getFaceValue();
        System.out.println("Sum: " + sum);
        sum = die1.roll() + die2.roll();
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);
        System.out.println("New sum: " + sum);
    }
}

Sample Run
Die One: 5, Die Two: 2
Die One: 1, Die Two: 4
Sum: 5
Die One: 4, Die Two: 2
New sum: 6

Another Example:
RollingDice.java

//********************************************************************
// Die.java  Author: Lewis/Loftus
// Represents one die (singular of dice) with faces showing values
// between 1 and 6.
//********************************************************************
public class Die
{
    private final int MAX = 6; // maximum face value
    private int faceValue;
    // current value showing on the die

    // Constructor: Sets the initial face value.
    public Die()
    {
        faceValue = 1;
    }

    // Rolls the die and returns the result.
    public int roll()
    {
        faceValue = (int)(Math.random() * MAX) + 1;
        return faceValue;
    }

    // Face value mutator.
    public void setFaceValue(int value)
    {
        faceValue = value;
    }

    // Face value accessor.
    public int getFaceValue()
    {
        return faceValue;
    }

    // Returns a string representation of this die.
    public String toString()
    {
        String result = Integer.toString(faceValue);
        return result;
    }
}

Next: Focus on Method definition
- parameters
- return type
- return statement

char ch = obj.calc (start, 2, "ABCDE");
char calc (int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt (sum);
    return result;
}
Method Control Flow

- If the called method is in the same class, only the method name is needed.

```java
myMethod();
```

Method Control Flow

- The called method is often part of another class or object.
- Thus the dot operator is an addressing mechanism. Note that it can also be used to access an object’s or class’s data directly, for example:
  - acct1.name
  - Color.black
- more on this later (encapsulation)

Invoking methods within the same class

- An object’s method may access any of its other methods directly. Eg:

```java
public void addInterest(double rate) {
    deposit (rate*balance);
}
```

Invoking methods within the same class

- The main method can do this too!

```java
public static void main (String[] args) {
    printManyGreetings(5);
}
```

- assumes `printManyGreetings()` is defined in the same class
- ➔ A convenient way to test methods
Invoking methods within the same class

- The main method can do this too!

```java
public static void main (String[] args)
{
    printManyGreetings(5);
}
```

- assumes `printManyGreetings()` is defined in the same class

**NOTE:** Since `main()` is `static`, `printManyGreetings()` must also be `static`

Method Control Flow revisited

- Recall, if the called method is in the same class, only the method name is needed

Implementing `printManyGreetings()` method:

```java
Invoking the method:
printManyGreetings(5);
```

More Method Examples:

- Write a method with two `double` parameters `a` and `b` that computes and returns the sum of squares of its two parameters (i.e., \(a^2 + b^2\)).

How do we invoke the method to compute & print: \((14.8)^2 + (37.65)^2\)?
More Method Examples:

- Write a method with one int parameter num, that returns a String composed of “Happy Birthday” num times

How do we invoke the method to print “happy birthday” 4 times?

Getting to know classes so far

- Using predefined classes from the Java API.
- Defining classes for our own datatypes.

**datatypes:**
- Account
- Die
- Shoe
- Person

**Clients (Driver classes):**
- Transactions, OnePercent
- RollingDice
- YouVeGotShoes (Project)
- PeopleBeingPeople (Lab)

Next: Focus on method definition/invocation

UML Class Diagrams

UML = Unified Modelling Language

- Example: A UML class diagram for the RollingDice program:

```
RollingDice
- main (args : String[]) : void

Die
- faceValue : int
- roll() : int
- setFaceValue(value : int) : void
- getFaceValue() : int
- toString() : String
```

UML class diagram for Transactions program?
Encapsulation

- An encapsulated object can be thought of as a black box – its inner workings are hidden from the client.
- The client invokes the interface methods which in turn manage the instance data.

**Use Accessors & Mutators - RIGHT**

- Indirect access through methods:
  - accessors and mutators (“getters” and “setters”)
  - Usually named `getX()` and `setX()`

**Visibility Modifiers**

- In Java, we enforce encapsulation through the appropriate use of visibility modifiers:
  - `public` – can be referenced from other classes
  - `private` – can be referenced only within that class:
    - `protected` – involves inheritance (discussed later)
- Data declared without a visibility modifier have default visibility and can be referenced by any class in the same package.
- An overview of all Java modifiers is presented in Appendix E.

**Violating Encapsulation - WRONG**

- It is possible for a class to access the instance data of another class directly – *but it's not a good idea!*
- See `Account.java`
- See `ImInUrClassMessingUrInstanceData.java`

**Example**

```java
int x1 = acct1.getBalance();
```

**Violating Encapsulation**

It is possible for a class to access the instance data of another class directly.
Violating Encapsulation experiment

- Revisit your solution for the [Account Class Exercise](#).
- Add some code to the OnePercent.java class to modify the value of an instance variable, e.g:
  ```java
cacct1.name = "Joe";
```
- This should work as expected.
- Now modify Account.java – insert the modifier `private` in front of that variable declaration:
  ```java
private String name;
```
- Re-compile the Account class and run your program again. Note the error you get.

public constants are ok

Example: The Account class can have a constant for the interest rate:
```java
public final double RATE = 0.015;
```
A client (e.g., OnePercent.java) can access this constant directly:
```java
System.out.print("Interest rate = " + acct1.RATE);
```

public constants are ok

Example: The Account class can have a constant for the interest rate:
```
Usually, constants are declared `static`:
```
```java
public final static double RATE = 0.015;
```
A driver class (e.g., OnePercent.java) can access this constant directly **without creating an object**:
```java
System.out.print("Interest rate = " + acct1.RATE);
```
```java
System.out.print("Interest rate = " + Account.RATE);
```

Visibility Modifiers – the **RULES**

<table>
<thead>
<tr>
<th>Variables</th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>(but OK for public constants)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>Yes, for support methods only</td>
</tr>
</tbody>
</table>

---