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
  - Account()
  - deposit()
  - withdraw()
  - getBalance()
  - toString()

Account object

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

The object:
- is like the house built from the blueprint
- is an instance of the class
- has its own data space & shares methods defined for this datatype

Classes define DATA and METHODS i.e., a datatype

2. Object instantiation

Creating Objects – OLD example:

- We have already seen something like this:

```
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:

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

Invokes the Account constructor, which is a special method that sets up the object

A new Account object is created!

acct1

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

3. Method invocation

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

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

```
acct1

acctNumber: 72354
balance: 128.41
name: "Ted Murphy"
```
Continue

---

Transactions.java

/// Account.java  Author: Lewis/Loftus
/// Represents a bank account with methods deposit and withdraw.
/// Returns a one-line description of the account as a string.

import java.text.NumberFormat;

public class Account {
    int acctNumber;
    double balance;
    String name;

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

    public double getBalance () {
        return balance;
    }

    public String toString () {
        return acctNumber + name + balance + NumberFormat.getCurrencyInstance().format(balance);
    }

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

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

    public void getBalance () {
    }

    public String toString () {
        return acctNumber + name + balance + NumberFormat.getCurrencyInstance().format(balance);
    }

}

public static void main (String[] args) {
    Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
    Account acct2 = new Account ("Jane Smith", 93705, 150.32);
    Account acct3 = new Account ("Edward Demsey", 93705, 759.32);
    System.out.println (acct1);
    System.out.println (acct2);
    System.out.println (acct3);
    acct1.deposit (25.85);
    acct2.deposit (125.00);
    acct3.deposit (45.00);
    acct1.withdraw (60, 2.50);
    acct2.withdraw (120.00, 2.50);
    acct3.withdraw (125.00, 2.50);
    acct1.getBalance () ;
    acct2.getBalance () ;
    acct3.getBalance () ;
    System.out.println (acct1.toString());
    System.out.println (acct2.toString());
    System.out.println (acct3.toString());
}

---

Transactions.java

/// Transactions.java  Author: MA Papalaskari
/// Demonstrates the creation and use of multiple Account objects.
///

public class Transactions {
    public static void main (String[] args) {
        Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
        Account acct2 = new Account ("Jane Smith", 93705, 150.32);
        Account acct3 = new Account ("Edward Demsey", 93705, 759.32);
        System.out.println (acct1);
        System.out.println (acct2);
        System.out.println (acct3);
        acct1.deposit (25.85);
        acct2.deposit (125.00);
        acct3.deposit (45.00);
        acct1.withdraw (60, 2.50);
        acct2.withdraw (120.00, 2.50);
        acct3.withdraw (125.00, 2.50);
        acct1.getBalance () ;
        acct2.getBalance () ;
        acct3.getBalance () ;
        System.out.println (acct1.toString());
        System.out.println (acct2.toString());
        System.out.println (acct3.toString());
    }

}
Transactions.java

Author: MA Papalaskari

// Demonstrates the creation and use of multiple Account objects.

public class Transactions {
    public static void main (String[] args) {
        Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
        Account acct2 = new Account ("Jane Smith", 69713, 40.00);
        Account acct3 = new Account ("Edward Demsey", 93757, 759.32);
        System.out.println (acct1);
        System.out.println (acct2);
        System.out.println (acct3);
        acct1.deposit (25.85);
        acct1.withdraw (60, 2.50);
        System.out.println ();
        System.out.println (acct1.toString());
        System.out.println (acct2.toString());
        System.out.println (acct3.toString());
    }
}

Transactions class:
Creating Account objects

Transactions class:
Creating more Account objects

Transactions class:
Account class: Using methods

---

Sample Run

<table>
<thead>
<tr>
<th>Account</th>
<th>Name</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>acct1</td>
<td>Ted Murphy</td>
<td>$102.56</td>
</tr>
<tr>
<td>acct2</td>
<td>Jane Smith</td>
<td>$40.00</td>
</tr>
<tr>
<td>acct3</td>
<td>Edward Demsey</td>
<td>$759.32</td>
</tr>
</tbody>
</table>

acct1.deposit (25.85):

// Deposits the specified amount into the account.

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);

acct1.deposit (25.85);

acct1.withdraw (60, 2.50);
Account class: Using methods

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

---

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

---

Account class: Another Example

```java
acct1.withdraw (60, 2.50);
```

---

// Withdraws the specified amount from the account and applies the fee.
public void withdraw (double x, double fee)
{
    balance = balance - x - fee;
}

---

getBalance () method

```java
double amount = acct1.getBalance();
```

---

// Note: this code is not used in Transactions.java

class
{
    public double getBalance ()
    {
        return balance;
    }
}
**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)

**toString() method**

```java
public String toString ()
{
    NumberFormat fmt = NumberFormat.getCurrencyInstance ();
    return acctNumber + "\n" + name + "\n" + fmt.format (balance);
}
```

**Can be omitted!**

**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)

**toString() method**

```java
public String toString ()
{
    NumberFormat fmt = NumberFormat.getCurrencyInstance ();
    return acctNumber + "\n" + name + "\n" + fmt.format (balance);
}
```

**Can be omitted!**

**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)

**toString() method**

```java
public String toString ()
{
    NumberFormat fmt = NumberFormat.getCurrencyInstance ();
    return acctNumber + "\n" + name + "\n" + fmt.format (balance);
}
```

**Can be omitted!**

**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)

**toString() method**

```java
public String toString ()
{
    NumberFormat fmt = NumberFormat.getCurrencyInstance ();
    return acctNumber + "\n" + name + "\n" + fmt.format (balance);
}
```

**Can be omitted!**

**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)

**toString() method**

```java
public String toString ()
{
    NumberFormat fmt = NumberFormat.getCurrencyInstance ();
    return acctNumber + "\n" + name + "\n" + fmt.format (balance);
}
```

**Can be omitted!**
NEXT: Focus on **Methods**

- Method control flow
- Method definition
  - Parameters
  - *return* statement
- Different ways of thinking about methods

Method Control Flow

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

```
// 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;
// ------------------------------------------------------
```

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 the object’s other methods directly. Eg:

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

Client code, eg: compound the interest for acct2 over 10 years

```java
int year = 1;
while (year <= 10) {
    acct2.addInterest(0.03);
    year ++;
}
```

Method definition: Example

– parameters
– return type
– return statement

```java
char ch = obj.calc(start, 2, "ABCDE");
```

```java
char calc(int num1, int num2, String message) {
    int sum = num1 + num2;
    char result = message.charAt(sum);
    return result;
}
```

Method Control Flow: example

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., a² + b²).

```java
char ch = obj.calc(start, 2, "ABCDE");
```

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.

<table>
<thead>
<tr>
<th>datatypes:</th>
<th>Clients (Driver classes):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>Transactions, OnePercent</td>
</tr>
<tr>
<td>Die</td>
<td>RollingDice</td>
</tr>
<tr>
<td>Shoe</td>
<td>YouVeGotShoes (Project)</td>
</tr>
<tr>
<td>Person</td>
<td>PeopleBeingPeople (Lab)</td>
</tr>
</tbody>
</table>

Next: UML class diagrams, visibility modifiers, graphical objects,

UML Class Diagrams

UML = Unified Modelling Language

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

\[ \text{RollingDice} \rightarrow \text{Die} \]

- `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

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

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 experiment

- Revisit the Account example (use our simplified versions from the course website or Lab 8)
- Add some code to the client (Transactions or Onepercent) to modify the value of an instance variable, eg:
  ```java
cacct1.name = "Bernie";
```
- Run the program to verify that the name on that account has changed.
- Now modify Account.java -- insert the modifier `private` in front of the instance variable declaration:
  ```java
private String name;
```
- Without changing anything in the client, re-compile the Account class and run your program again. Note the error you get:

```
Error: ____________________________________________________
```

Public Constants… OK

Example: The Account class can have a constant for the interest rate:

```java
public final double RATE = 0.015;
```

A client (eg, OnePercent.java) can access this constant directly:

```java
System.out.print("Interest rate = " + acct1.RATE);
```

**static** Public Constants… BETTER

It is better to declare constants as `static`

```java
public final static double RATE = 0.015;
```

This way, a client can access the constants without creating an object, using the class name:

```java
System.out.print("Interest rate = " + Account.RATE);
```
Visibility Modifiers – the **RULES**

<table>
<thead>
<tr>
<th></th>
<th><code>public</code></th>
<th><code>private</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variables</strong></td>
<td>NO</td>
<td>Yes</td>
</tr>
<tr>
<td>(but OK for public constants)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td>Yes</td>
<td>Yes, for support methods only</td>
</tr>
</tbody>
</table>

See also [ImInUrClassMessingUrInstanceData.java](#)

Graphical Objects

- Some objects contain information that determines how the object should be represented visually
- **Graphical objects**
  - data about position, size, and other attributes
  - methods to draw the object
- Let’s look at some other examples of graphical objects:
  - **Example 1:** SmilingFacePanel
    - SmilingFace.java
    - SmilingFacePanel.java
  - **Example 2:** SplatPanel
    - Splat.java
    - SplatPanel.java
    - Circle.java

```java
// SmilingFace.java  Author: Lewis/Loftus
// Demonstrates the use of a separate panel class.
//**************************************************
import javax.swing.JFrame;
public class SmilingFace
{
  // Creates the main frame of the program.
  public static void main (String[] args)
  {
    JFrame frame = new JFrame("Smiling Face");
    frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
    SmilingFacePanel panel = new SmilingFacePanel();
    frame.getContentPane().add(panel);
    frame.pack();
    frame.setVisible(true);
  }
}
```
The SmilingFace program draws a face by defining the paintComponent method of a panel.

A UML class diagram:

```
SmilingFace
  main (args : String[]): void

SmilingFacePanel
  BASEX: int
  BASEY: int
  paintComponent(p: Graphics): void
```

The SmilingFacePanel class is derived from the JPanel class using inheritance. We are defining a subclass of JPanel.
Smiling Face Example

- Every Swing component has a `paintComponent` method
- The `paintComponent` method accepts a `Graphics` object that represents the graphics context for the panel
- We define the `paintComponent` method to draw the face with appropriate calls to the `Graphics` methods
- Note the difference between drawing on a panel and adding other GUI components to a panel

Objects with a `draw()` method

- The next example - Splat - is structured differently
- It draws a set of colored circles on a panel, but each circle is represented as a separate object that maintains its own graphical information
- The `paintComponent` method of the panel "asks" each circle to draw itself
- See `Splat.java`
- See `SplatPanel.java`
- See `Circle.java`

```java
import javax.swing.*;
import java.awt.*;
public class Splat {
    public static void main (String[] args) {
        JFrame frame = new JFrame("Splat");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(new SplatPanel());
        frame.pack();
        frame.setVisible(true);
    }
}
```
SplatPanel.java

import javax.swing.*;
import java.awt.*;

public class SplatPanel extends JPanel {
  private Circle circle1, circle2, circle3, circle4, circle5;

  public SplatPanel() {
    circle1 = new Circle(30, Color.red, 70, 35);
    circle2 = new Circle(50, Color.green, 30, 20);
    circle3 = new Circle(100, Color.cyan, 60, 85);
    circle4 = new Circle(45, Color.yellow, 170, 30);
    circle5 = new Circle(60, Color.blue, 200, 60);

    setPreferredSize(new Dimension(300, 200));
    setBackground(Color.black);
  }

  public void paintComponent(Graphics page) {
    super.paintComponent(page);
    circle1.draw(page);
    circle2.draw(page);
    circle3.draw(page);
    circle4.draw(page);
    circle5.draw(page);
  }
}

continue
import java.awt.*;
public class Circle
{
    private int diameter, x, y;
    private Color color;

    public Circle (int size, Color shade, int upperX, int upperY)
    {
        diameter = size;
        color = shade;
        x = upperX;
        y = upperY;
    }

    public void draw (Graphics page)
    {
        page.setColor (color);
        page.fillOval (x, y, diameter, diameter);
    }

    public void setDiameter (int size)
    {
        diameter = size;
    }

    public void setColor (Color shade)
    {
        color = shade;
    }

    public void setX (int upperX)
    {
        x = upperX;
    }

    public void setY (int upperY)
    {
        y = upperY;
    }

    public int getDiameter ()
    {
        return diameter;
    }

    public Color getColor ()
    {
        return color;
    }

    public int getX ()
    {
        return x;
    }

    public int getY ()
    {
        return y;
    }
}