CS 106A, Lecture 14
Classes and Objects

reading:

Art & Science of Java, Chapter 6

Based on slides created by Keith Schwarz, Mehran Sahami, Eric Roberts, Stuart Reges, and others.

Lecture at a glance

- Today we will learn to create **classes** of **objects**.
  - Writing a class defines a new data type.
  - Classes are crucial for writing large Java applications.

- Examples:
  - A calendar program might want a **`Date`** class.
  - A student registration system might want a **`Student`** class.
  - A bank app might want a **`BankAccount`** class.

Classes and objects

- **class**: A program entity that represents either:
  1. A program / module, or
  2. **A template for a new type of objects.**

- **object**: An entity that combines state and behavior.
  - Example: You can create `Student` objects using the `Student` class.
  - Each object is also called an *instance* of a class.

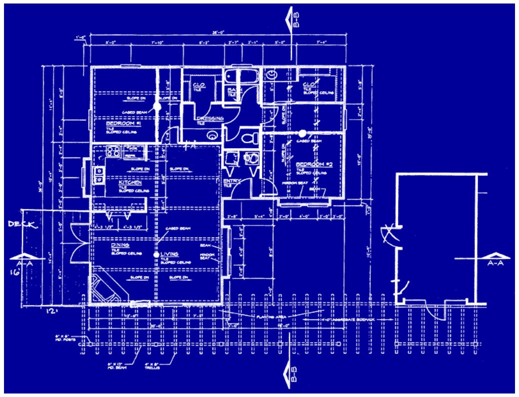
- **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.

Blueprint analogy

iPod blueprint (class)

state:
current song
volume
battery life

behavior:
power on/off
change station/song
change volume
choose random song



constructs

iPod (object) #1

state:
song = "1,000,000 Miles"
volume = 17
battery life = 2.5 hrs

behavior:
power on/off
change station/song
change volume
choose random song

iPod (object) #2

state:
song = "Letting You"
volume = 9
battery life = 3.41 hrs

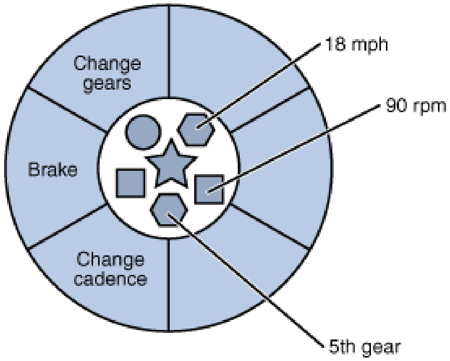
behavior:
power on/off
change station/song
change volume
choose random song

iPod (object) #3

state:
song = "Discipline"
volume = 24
battery life = 1.8 hrs

behavior:
power on/off
change station/song
change volume
choose random song

Elements of a class



- **fields**: State (variables) inside each object.
  - Declared as `private`
  - Each object created has a copy of each field.
- **methods**: Behavior (code) that executes inside each object.
  - Each object created has a copy of each method.
  - The method can interact with the data inside that object.
- **constructor**: Initializes new objects as they are created.
  - Sets the initial state of each new object.
  - Often accepts parameters for the initial state of the fields.

Fields

- **field**: A variable inside an object that is part of its state.
  - Each object gets *its own copy* of each field.

- Declaration syntax:

```
private type name;
```

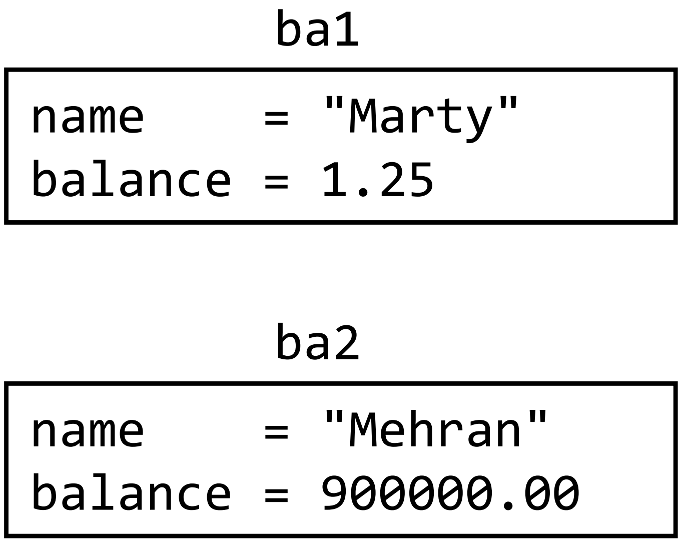
  - access level is typically `private`, but can be `public` or others

  - Example:

```
public class BankAccount {
    private String name;      // each account object has
    private double balance;   // a name and balance field
    ...
```

Using objects

```
BankAccount ba1 = new BankAccount();
ba1.name = "Marty";
ba1.balance = 1.25;

BankAccount ba2 = new BankAccount();
ba2.name = "Mehran";
ba2.balance = 900000.00;
```



```
ba1
name    = "Marty"
balance = 1.25

ba2
name    = "Mehran"
balance = 900000.00
```

- Think of an object as a way of grouping multiple variables.
  - Each object contains a name and balance field inside it.
  - We can get/set them individually.
  - Code that uses your objects is called *client* code.

Instance methods

- **instance method** (or **object method**): Exists inside each object of a class, and gives behavior to each object.

```
public type name(parameters) {
    statements;
}
```

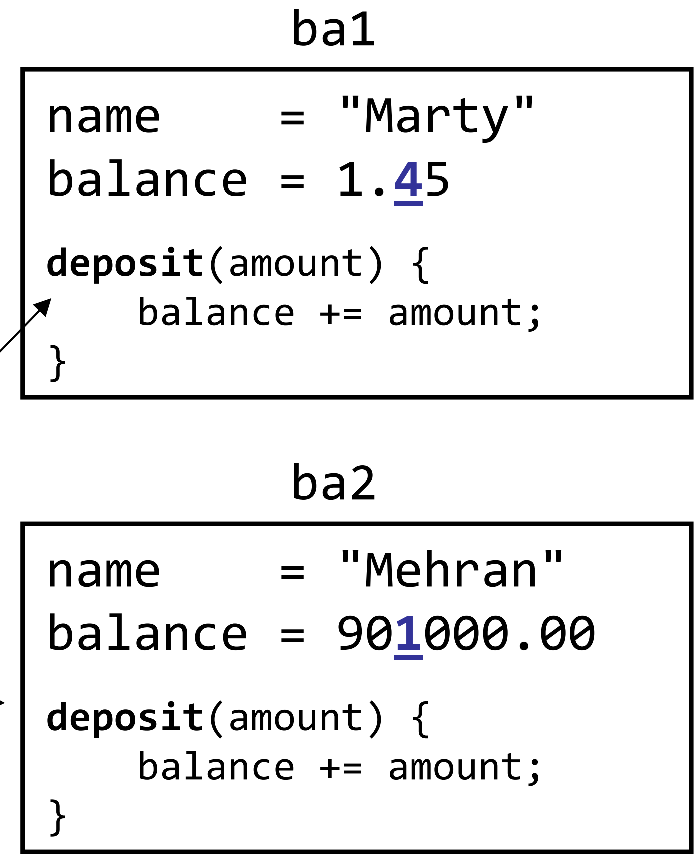
  - access level is usually `public`, but can be `private` or others

  - Example (in `BankAccount` class):

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

Using object methods

```
BankAccount ba1 = new BankAccount();
ba1.name = "Marty";
ba1.balance = 1.25;

BankAccount ba2 = new BankAccount();
ba2.name = "Mehran";
ba2.balance = 900000.00;

ba1.deposit(0.20);
ba2.deposit(1000.00);
```



ba1

```
name    = "Marty"
balance = 1.45

deposit(amount) {
    balance += amount;
}
```

ba2

```
name    = "Mehran"
balance = 901000.00

deposit(amount) {
    balance += amount;
}
```

- When you call an object's method:
  - It executes that object's copy of the code from the class.
  - If that code refers to fields, it means that object's copy of those fields.
  - So calling the method on different objects has different effects.

The implicit parameter

- **implicit parameter**:
  The object on which an instance method is called.
  - If the client makes the call, `ba1.deposit(20.00);`
    the object named `ba1` is the implicit parameter for that call.
  - If the client makes the call, `ba2.deposit(20.00);`
    the object named `ba2` is the implicit parameter for that call.
  - KEY POINT: An instance method can directly access the fields of the object on which it was called.
    - We say that it executes in the *context* of a particular object.
    - `deposit` can refer to the `name`, `balance` of the account it was called on

Usefulness of methods

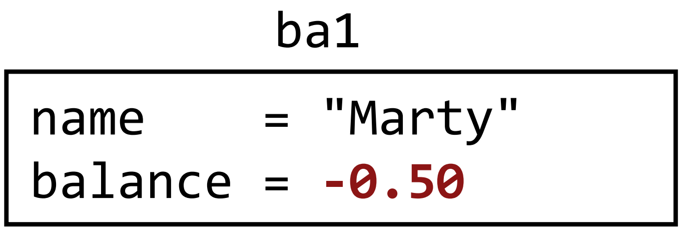
- Having methods like **`deposit`** and **`withdraw`** to change the balance is better than having the client modify it directly.
  - Can enforce *invariant* constraints like, "no negative balances", etc.

```
// Adds the given amount of money to the account.
// If the amount is negative, has no effect.
public void deposit(double amount) {
    if (amount > 0.0) {
        balance += amount;
    }
}

// Deducts the given amount of money from the account.
// If the amount is negative or > balance, no effect.
public void withdraw(double amount) {
    if (amount > 0.0 && amount <= balance) {
        balance -= amount;
    }
}
```

Inappropriate access

- If client code could bypass our `deposit` and `withdraw` methods, it could set the balance of an account to an invalid value.

```
BankAccount ba1 = new BankAccount();
ba1.name = "Marty";
ba1.balance = 1.50;

// Haha, I bypassed your method!
ba1.balance -= 2.00;
```



```
           ba1
name    = "Marty"
balance = -0.50
```

  - This is bad; it violates our invariant of, "balance is never negative".
  - How can we stop bad clients from doing this?

Private fields

A field that cannot be accessed from outside the class (.java file)

```
private type name;
```

- Examples:

```
private String name;
```

Initializing objects

- If fields are private, we can't directly initialize a BankAccount:

```
BankAccount ba1 = new BankAccount();
ba1.name = "Marty";
ba1.balance = 0.50;   // no longer compiles
```

- Client code will not compile if it tries to access private fields:

```
BankClient.java:27: name has private access in BankAccount
ba1.name = "Marty";
    ^
```

- How can we enable the following syntax in our class?

```
BankAccount ba1 = new BankAccount("Marty", 0.50);   // better!
```

Constructors

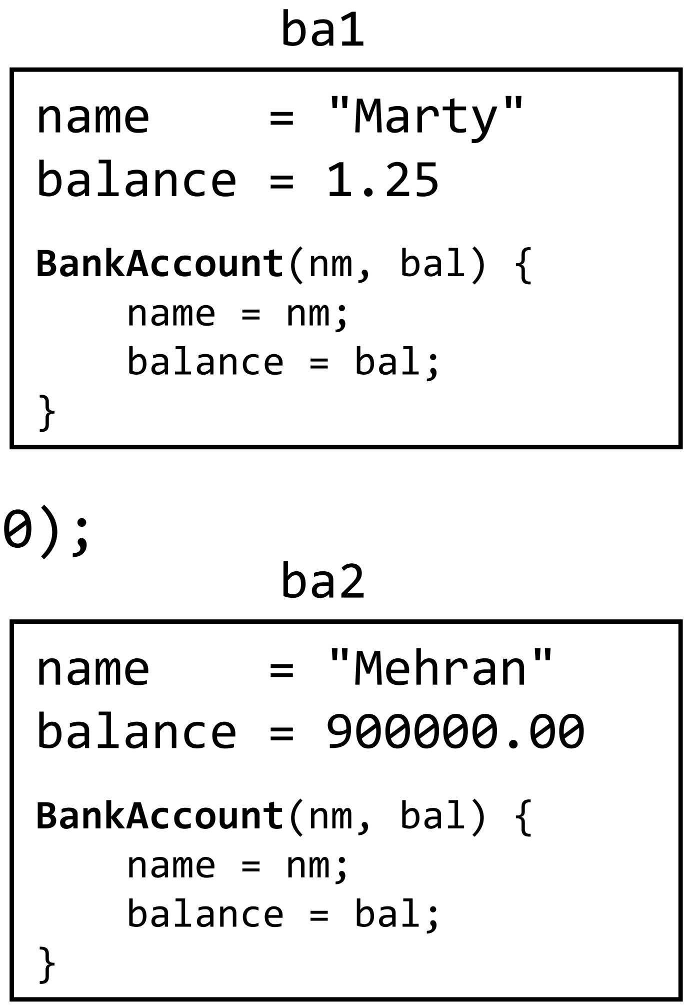
- **constructor**: Initializes the state of new objects as they are created.

```
public ClassName(parameters) {
    statements;
}
```

  - The constructor runs when the client says `new ClassName(...);`
  - no return type is specified; it "returns" the new object being created
  - If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all fields to default values like `0` or `null`.

Using constructors

```
BankAccount ba1 =
    new BankAccount("Marty", 1.25);
```



ba1

```
name    = "Marty"
balance = 1.25

BankAccount(nm, bal) {
    name = nm;
    balance = bal;
}
```

```
BankAccount ba2 =
    new BankAccount("Mehran", 900000.00);
```

ba2

```
name    = "Mehran"
balance = 900000.00

BankAccount(nm, bal) {
    name = nm;
    balance = bal;
}
```

- When you call a constructor (with new):
  - Java creates a new object of that class.
  - The constructor runs, with that new object as the implicit parameter.
  - The newly created object is returned to your program.

BankAccount exercises

- Write a method **`setTransactionFee`** that incurs a fee every time the client deposits or withdraws from that account.
  - Example: if you set transaction fee to $0.50 and then withdraw $8.00, then $8.50 is actually withdrawn.
  - Make sure an account cannot withdraw more than (amount + fee).

- Write a method **`printLog`** that shows all transactions so far.
  - Make each account keep an internal log String of all transactions.
  - Example output from `printLog` :

```
Deposit of $7.82
Withdrawal of $2.55
Deposit of $6.18
```

Printing objects

- By default, Java doesn't know how to print objects.

```
// ba1 is BankAccount@9e8c34
BankAccount ba1 = new BankAccount("Marty", 1.25);
println("ba1 is " + ba1);

// better, but cumbersome to write
// ba1 is Marty with $1.25
println("ba1 is " + ba1.getName() + " with $"
        + ba1.getBalance());


// desired behavior
println("b1 is " + ba1);   // ba1 is Marty with $1.25
```

The toString method

tells Java how to convert an object into a string

```
BankAccount ba1 = new BankAccount("Marty", 1.25);
println("ba1 is " + ba1);

// the above code is really calling the following:
println("ba1 is " + ba1.toString());
```

- Every class has a `toString`, even if it isn't in your code.
  - Default: class's name @ object's memory address (base 16)

```
BankAccount@9e8c34
```

toString syntax

```
public String toString() {
    code that returns a String
    representing this object;
}
```

- Method name, return, and parameters must match exactly.

- Example:

```
// Returns a String representing this account.
public String toString() {
    return name + " has $" + balance;
}
```

The keyword this

- **`this`** : A reference to the implicit parameter.
  - *implicit parameter:* object on which a method is called

- Syntax for using `this` :
  - To refer to a field:

    `this.`***`field`***

  - To call a method:

    `this.`***`method`***`(`***`parameters`***`);`

  - To call a constructor from another constructor:

    `this(`***`parameters`***`);`

Variable names/scope

- Usually illegal to have 2 variables in same scope with same name:

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String newName) {
        name = newName;
    }
}
```

  - The parameter to `setName` is named newName to be distinct from the object's field name .

Variable shadowing

- An instance method parameter name can match a field name:

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String name) {
        this.name = name;
    }
}
```

  - Field name is *shadowed* by the parameter with the same name.
  - Any code inside `setName` that refers to `name` will use the parameter, not the field. To refer to the field, say `this.name` .

Multiple constructors

- It is legal to have more than one constructor in a class.
  - The constructors must accept different parameters.

```
public class BankAccount {
    private double balance;
    private String name;

    public BankAccount(String name) {
        this.name = name;
        balance = 0.00;
    }

    public BankAccount(String name, double bal) {
        this.name = name;
        balance = bal;
    }
    ...
}
```

Multiple constructors

- One constructor can call another using `this` :

```
public class BankAccount {
    private double balance;
    private String name;

    public BankAccount(String name) {
        this(name, 0.00); // call other constructor
    }

    public BankAccount(String name, double bal) {
        this.name = name;
        balance = bal;
    }
    ...
}
```