Encapsulation

Mason Vail
Boise State University Computer Science

Pillars of Object-Oriented Programming

- **Encapsulation**
- Inheritance
- Polymorphism
- Abstraction (sometimes)

Object Identity

Data (variables) make one object different from another.

An object’s data make up its identity.

Examples:

- Student: name, ID, address, major, GPA, etc.
- Book: title, author, copyright, edition, etc.
- BankAccount: accountNumber, owner, balance, etc.

Need for Protecting Data

Imagine your bank account is represented as an object in the bank's software.

There are specific valid ways that your account balance can be modified: deposits, withdrawals, fees, and interest.

Any other modification to an account balance other than through those transactions should not be allowed.

```
    for(Account acc : accounts) {
        acc.balance = 0;
    }
```

Encapsulation: Protecting Data through Visibility

Is it possible to guarantee that data can only be affected by valid operations?

Yes.

Encapsulation is the concept of protecting an object's data so that it can only be modified through acceptable, valid operations.

Encapsulation is enforced through use of **visibility modifier**s - keywords that specify where data and operations can be seen and used.

In Java, the primary distinction is between **public** and **private** visibility.

Encapsulation: Protecting Data through Visibility

Visibility modifiers should be explicitly applied to all instance (and class) variables, constants, and methods.

Visibility of all variables and all internal support methods should be private.

```
    private long accountNumber;
    private String accountOwner;
    private double balance;
```

Only methods *intended* to be called by outside users should be public.

```
    public void deposit(double amtToAdd) { … }
    public double getBalance() { … }
```

Access to private data, then, will be indirect, through public methods only.

Visibility and Scope

Variables, constants, or methods declared as *private* can be seen and used anywhere inside the scope of the class - in any code within the class. No code outside the class, however, can see or use anything declared as private for that class.

Anything declared *public* can be seen and used in code inside or outside the class. This is why it is important that a class only exposes a limited set of appropriate methods to the outside world. Public methods make up the public interface of an object of the class type, through which other code will interact with the object.

Accessor (“getter”) and Mutator (“setter”) Methods

Methods that return the current value of an instance variable or update an instance variable value are common.

A method that simply returns a current value is called an **accessor**, or “getter” method. A method that directly updates a current value is called a **mutator**, or “setter” method.

```
public int getSize() { return size; }
public void setSize(int newSize) { size = newSize; }
```

When writing a class, you only need to provide accessors or mutators for the variables you *want* a user to be able to view or update. Mutator methods often contain logic to validate any requested changes. For example, a mutator could confirm that a new value is within a required range.

Simple Encapsulated Class: BankAccount (part 1)

```
public class BankAccount {
  //all instance variables are private
  private long acctNumber;    //accessor below, but no mutator
  private String acctOwner;   //accessor below, but no mutator
  private double acctBalance; //accessor below, changes via deposit() and
withdraw()

  //constructor initializes instance variables
  public BankAccount(long acctNumber, String acctOwner) {
    this.acctNumber = acctNumber;
    this.acctOwner = acctOwner;
    This.balance = 0.0;
  }

  ...
}
```

Simple Encapsulated Class: BankAccount (part 2)

```
public class BankAccount {
  ...

  //public accessor methods return current values of instance variables
  public long getAcctNumber() {
    return acctNumber;
  }
  public String getAcctOwner() {
    return acctOwner;
  }
  public double getAcctBalance() {
    return acctBalance;
  }

  ...
}
```

Simple Encapsulated Class: BankAccount (part 3)

```
public class BankAccount {
  ...

  public void deposit(double amtToAdd) { //update only if valid input
    if (amtToAdd > 0) {
      acctBalance += amtToAdd;
    } else {
      throw new InvalidTransactionException();
    }
  }
  public void withdraw(double amtToWithdraw) { //update only if valid input
    if (amtToWithdraw > 0 && acctBalance >= amtToWithdraw) {
      acctBalance -= amtToWithdraw;
    } else {
      throw new InvalidTransactionException();
    }
  }
}
```

Encapsulation

Mason Vail
Boise State University Computer Science