

Java semantics: Classes



Saman Amarasinghe
Michael Ernst
MIT EECS



Outline

Modularity
Object-oriented programming
Subclasses
Method dispatch
Interfaces

MIT 6.170

Slide 2



Achieving Modularity

Reduce and order complexity

Using decomposition and abstraction

Decompose programs into units such that

Each part has independent requirements
Interactions are limited in complexity
It is clear how to implement and test each part
The parts can be combined to solve the original problem

This is necessary, but not sufficient

Dreaded system integration failure

MIT 6.170

Slide 3



Abstraction makes decomposition work

The design loop

Abstract to get a simpler problem
Decompose the simpler problem
Define the interface between modules
Apply process recursively to new problem

Purely top-down is an over-simplification

Often combine top-down and bottom-up

We will discuss this issue later

MIT 6.170

Slide 4



Object-oriented programming

Encapsulation, data hiding, access control

Prevent direct access to internal data structures, operations
Ensure consistency (representation invariant not violated)

Inheritance (code sharing)

Slightly different behavior requires only a little new code

Data-oriented view; communication by message-passing

Which is central: the functions or the data?
Not an emphasis of this course

Java supports all three concepts via classes

MIT 6.170

Slide 5



Bank account system

The system will represent bank accounts and transactions

A class representing (modeling) a transaction:


```
class Transaction {
    int amount;
    Date date;
    Transaction(int amount, Date date) {
        this.amount = amount;
        this.date = date;
    }
    Transaction(int amount) {
        this(amount, new Date()); // today
    }
}
```

Use of the class:

```
Transaction t = new Transaction(100);
```

MIT 6.170

Slide 6



A bank account


```

class Account {
    String name;
    List<Transaction> transactions
    = new ArrayList<Transaction>();
    int balance = 0;

    Account(String name) {
        this.name = name;
    }
    boolean hasSufficientFunds(Transaction t) {
        return (balance + t.amount) >= 0;
    }
    void post(Transaction t) {
        if (hasSufficientFunds(t)) {
            transactions.add(t);
            balance += t.amount;
        }
    }
}

```

MIT 6.170 Slide 7




Use of the class

```

Account copAcct = new Account("Copperfield");
Transaction t1 = new Transaction(100);
copAcct.post(t1);

```

MIT 6.170 Slide 8



Subclassing

```


class OverdraftAccount extends Account {
    int creditLimit;

    OverdraftAccount(String name, int limit) {
        super(name);
        creditLimit = limit;
    }

    boolean hasSufficientFunds(Transaction t) {
        return (balance + t.amount + creditLimit) >= 0;
    }
    void improveCredit(int amount) {
        creditLimit += amount;
    }
}

```

MIT 6.170 Slide 9



Use of the class


```

Account warbAcct
    = new OverdraftAccount("Warbucks", 500);
Transaction t2 = new Transaction(-100);
warbAcct.post(t2);

```

Which version of hasSufficientFunds is called?

MIT 6.170 Slide 10




The bank itself

```

class Bank {
    List<Account> accounts = new ArrayList<Account>();
    ...
    void chargeMonthlyFee() {
        for (Account acct : accounts) {
            Transaction fee = new Transaction(-1);
            if (acct.hasSufficientFunds(fee)) {
                acct.post(fee);
            }
        }
    }
    ...
}

```

MIT 6.170 Slide 11



Compile-time and run-time types

- Each **expression** (including fields) has a **declared type**
- Each **object** has an actual type at **run time**
- The declared type determines which operations the **compiler permits**
 - This guarantees that no "unknown method" or "unknown field" errors occur at run time
- The actual type determines **which implementation of the method is called**
- Casts have no effect on the object**
 - They perform a run-time check
 - They change the declared type
 - In Java 1.5, casts are usually an indication of bad style

MIT 6.170 Slide 12



Abstract classes and interfaces

An abstract class is partially implemented

- Some declared methods are not implemented
- The class cannot be instantiated
- Subclasses inherit implementations, fields
- A concrete subclass must implement all methods

An interface is not at all implemented

- Methods may be declared
- No fields, no constructor

These are useful when you want certain functionality, but do not wish to impose an implementation

Generally:

- Abstract class for closely related classes
- Interface for functionality not dependent on the class

We will return to this later in the term
