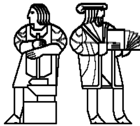
6.170 Lecture 3
Java semantics: Classes



Michael Ernst
MIT EECS

Outline

Modularity
Object-oriented programming
Subclasses
Method dispatch
Interfaces

Achieving Modularity

Reduce and order complexity
- Using decomposition and abstraction

Decompose programs into units such that
- Each has independent requirements
- Interactions limited in complexity
- Clear how to implement and test each
- Can be combined to solve original problem

Why is this not enough
- Dreaded system integration failure

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

Over-simplification
- Never strictly top-down

We will discuss this issue later

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 these via classes [and interfaces]

Bank account 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());
  }
}
```

> A constructor actually only initializes; the object already exists when the constructor is called.

> Use the same formal as the field name: because they are the same, and to improve documentation.

Use of the class:

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

A bank account

```
class Account {
  String name;
  Vector transactions = new Vector();
  int balance = 0;
  Account(String name) {
    this.name = name;
  }
  boolean check(Transaction t) {
    return (balance + t.amount) >= 0;
  }
  void post(Transaction t) {
    if (check(t)) {
      transactions.addElement(t);
      balance += t.amount;
    }
  }
}
```


Use of the class

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


Subclassing

```
class OverdraftAccount extends Account {
  int creditLimit;
  OverdraftAccount(String name, int limit) {
    super(name);
    creditLimit = limit;
  }
  boolean check(Transaction t) {
    return (balance + t.amount) >= - creditLimit;
  }
  void improveCredit(int amount) {
    creditLimit += amount;
  }
}
```


Use of the class

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

Which version of `check` is called?

The **OverdraftAccount** version, because the receiver is of that type, even though the `post` method is in **Account**.

The bank itself (example of casting)

```
class Bank {
  Vector accounts = new Vector();
  ...
  void chargeMonthlyFee() {
    for (int i=0; i<accounts.size(); i++) {
      Transaction fee = new Transaction(-1);
      Account acct = (Account) accounts.elementAt(i);
      if (acct.checkTrans(fee)) {
        acct.post(fee);
      }
    }
  }
  ...
}
```

The compile-time type of "accounts.elementAt(i)" is `Object`

The compile-time type of "(Account) accounts.elementAt(i)" is **Account**

The runtime type is unaffected by the cast

Michael Ernst:

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 runtime
- **The actual type determines which implementation of the method is called** This is called dispatching
- **Casts have no effect on the object**
  - They perform a runtime check
  - They change the declared type

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