OO concepts

UML representation

- Objects, Classes, Messages/Methods
- Inheritance, Polymorphism, Dynamic Binding
- Abstract Classes, Abstract Coupling
Lecture notes at:

http://www.softwareresearch.net/index.php?id=220
Objects in UML

- Object notation

An object diagram provides a runtime snapshot of the system, representing objects and the connections between them.
Object diagram
Class relationships (I)

An association can be refined by other relations

Often one models first only the fact that two classes are related and refines later this general notation element.
Class relationships (II)

- Each association can be named with a text label (like in the ER-model)
- Role names can be specified at association ends
- Multiplicity can be marked at association ends
- A class can have an association with itself, expressing a relationship between objects of the same class
Class relationships (III)

Multiplicity specification:

- 1: exactly one
- *: any (0 or more)
- 0..*: any (0 or more)
- 1..*: 1 or more
- 0..1: 0 or 1
- 2..5: range of values
- 1..5, 9: range of values or nine
Class relationships (IV)

Example:
Inheritance
Polymorphism
Dynamic Binding
Inheritance (I)

- A class defines the type of an object

- If one models for example a class `Customer` and a class `CorporateCustomer`, one expects that each object of type `CorporateCustomer` to be also of type `Customer`. The type `CorporateCustomer` is a subtype of `Customer`. 
Inheritance (II)

- A superclass generalizes a subclass
- A subclass specializes a superclass
- A subclass **inherits** methods and attributes of its superclass
A subclass has the following possibilities to specialize its behavior:

- Defining new operations and attributes
- Modifying existing operations (overwriting methods of the superclass)

Flatten view:
Inheritance (IV)

- UML Notation

```
Customer

+checkRegularCustomer():boolean

PrivateCustomer

+checkRegularCustomer():boolean

CorporateCustomer

+checkRegularCustomer():boolean
+report(doc:ActivityReport):void
```
Inheritance (V)

„delta“ view

Flatten view (not in standard UML!)

```
C1
- A1:int
+ m1():void

C2
- A2:int
```

```
C1
- A1:int

C2
- A1:int
- A2:int
+ m1():void
```
Inheritance and access rights

- Private members of a superclass are **not accessible** in subclasses
- Protected members of a superclass are accessible **only** in subclasses
- Public members are accessible **everywhere**
- Access rights can be specified globally for a superclass (C++):

  ```cpp
  class R : private A{ /* ... */ };
  class S : protected A{ /* ... */ };
  class T : public A{ /* ... */ };
  ```
Inheritance in Java

- Java supports single inheritance, where each class has at most one superclass
- The keyword is `extends`

Example:

```java
public class CorporateCustomer extends Customer{
    ...
}
```
Inheritance in C++

class Base {

    protected: int i;

};

class Derived : public Base {

    int f(Base* b) { return b->i; }

    int g(Derived* d) { return d->i; }

};
Inheritance
Polymorphism
Dynamic Binding
Polymorphism (I)

- An object type can be poly (=multiple) morph (=form). This can be depicted in the same way as plug-compatibility:

[Diagram showing objects compatible with the plug and "Plug"-Standard]
Inheritance example revisited

- **Customer**
  - `+checkRegularCustomer() : boolean`

- **PrivateCustomer**
  - `+checkRegularCustomer() : boolean`

- **CorporateCustomer**
  - `+checkRegularCustomer() : boolean`
  - `+report(doc: ActivityReport) : void`
Objects of type CorporateCustomer (subclass) keep at least the same contract as objects of type Customer (superclass).

Therefore it is meaningful to consider that an object of class $A_i$, which is a subclass of class $A$, is not only of type $A_i$ but also of the types given by all $A_i$‘s superclasses (starting with $A$).

An object has not only one type. It has multiple types, and the number of types is given by the position of the class from which the object is generated in the class hierarchy.
Polymorphism – Example (I)

```java
Customer customer = new Customer();
PrivateCustomer privateCustomer = new PrivateCustomer();
CorporateCustomer corporateCustomer = new CorporateCustomer();
```

![Diagram showing polymorphism example]

- Customer
  - checkRegularCustomer()

- PrivateCustomer
  - checkRegularCustomer()

- CorporateCustomer
  - checkRegularCustomer()
  - report(ActivityReport)
Polymorphism – Example (II)

customer = privateCustomer;  // OK

Customer
checkRegularCustomer()

PrivateCustomer
checkRegularCustomer()

CorporateCustomer
checkRegularCustomer()
report(ActivityReport)
customer = corporateCustomer;  // OK
Polymorphism – Example (IV)

privateCustomer = customer;    // wrong

```
    Customer
    checkRegularCustomer()
    
    PrivateCustomer
    checkRegularCustomer()
    
    CorporateCustomer
    checkRegularCustomer()
    report(ActivityReport)
```

```
customer
privateCustomer
```

```
corporateCustomer
```
corporateCustomer = customer;  // wrong

```
Customer

checkRegularCustomer()

PrivateCustomer

checkRegularCustomer()

CorporateCustomer

checkRegularCustomer()
report(ActivityReport)
```
Polymorphism – Example (VI)

- The reason for failure is that an object which is an instance of class `Customer` does not understand all method calls that an object which is an instance of class `CorporateCustomer` understands.

1) `corporateCustomer = customer;`
2) `corporateCustomer.report(monthlyReport);`

1) Type mismatch: cannot convert from `CorporateCustomer` to `Customer`
2) The method `report(activityReport)` is undefined for the type `Customer`. 
Polymorphism – Example (VII)

```
Person
  +getBirthDate():String
  +getHealthHistory():String

Hotel
  +Customer[]
  +addCustomer(Customer):void

Investor

Company
  +getDevelopmentPlan():Plan

PrivateCustomer
  +checkRegularCustomer():boolean

CorporateCustomer
  +checkRegularCustomer():boolean
  +report(doc:ActivityReport):void

Hotel
  +PrivateCustomer[]
  +CorporateCustomer[]
  +addPrivateCustomer(PrivateCustomer):void
  +addCorporateCustomer(CorporateCustomer):void
```
Static and dynamic type

- **Static type**
  - Accurately given by the declaration in the program text
  - Example: `customer` is of static type `Customer`

- **Dynamic type**
  - The type of the referenced object at runtime
  - Example: after `customer=corporateCustomer`, the dynamic type of `customer` is `CorporateCustomer`

- A variable with a static type can have several dynamic types during its lifetime, depending on the width and depth of the class hierarchy
Dynamic binding (I)

Dynamic binding: The compiler does not specify which method is called at runtime. The method is determined at runtime based on

- The method name
- The variable’s dynamic type

```java
Customer c;
if (i > 0) then
    c = new CorporateCustomer();
else
    c = new PrivateCustomer();
...
c.checkRegularCustomer();
```
Dynamic binding (II)

When \( i > 0 \) is true, the variable \( c \) references an object generated from the class `CorporateCustomer` (and thus has the dynamic type `CorporateCustomer`). Hence, the call to `checkRegularCustomer()` is linked to the method as implemented in `CorporateCustomer`.

- In Java, all methods are dynamically bound, except for the ones explicitly marked by using the keyword `static`.

- In C++, by contrast, methods must be explicitly marked as dynamically bound by using the keyword `virtual`.
Dynamic binding can be used for the plug-in concept

For example, the yellow object may implement \texttt{m1()} differently than the red object.
Inheritance exercise

```java
public class BaseTest {
    protected int protMember;
    BaseTest(int i) {
        protMember = i;
    }
}

public class DerivedA extends BaseTest {
    DerivedA(int i) {
        super(i);
    }
    public void printProt(BaseTest bt) {
        System.out.println("Value in base class is "+ bt.protMember);
    }
    public void printProt(DerivedB db) {
        System.out.println("Value in derived class is "+ db.protMember);
    }
}

public class DerivedB extends BaseTest {
    DerivedB(int i) {
        super(i);
    }
}

public class Worker {
    DerivedA da;
    DerivedB db;
    BaseTest bt;
    public void work() {
        db = new DerivedB(2);
        da = new DerivedA(1);
        da.printProt(db);
        bt = db;
        da.printProt(bt);
    }
    public static void main(String[] args) {
        Worker wk = new Worker();
        wk.work();
    }
}
```
Inheritance exercise

```java
public class BaseTest {
    protected static int protMember;

    BaseTest(int i) {
        protMember = i;
    }
}

public class DerivedA extends BaseTest {
    DerivedA(int i) {
        super(i);
    }
}

public class DerivedB extends BaseTest {
    DerivedB(int i) {
        super(i);
    }
}

public class Worker {
    DerivedA da;
    DerivedB db;
    BaseTest bt;

    public void work() {
        db = new DerivedB(2);
        da = new DerivedA(1);
        da.printProt(db);
        bt = db;
        da.printProt(bt);
    }

    public static void main(String[] args) {
        Worker wk = new Worker();
        wk.work();
    }
}
```
The diamond problem

```
Animal myPet = new BestPet();
myPet.talk();
```

This problem does not occur in Java
Is-A and Has-A

Typical error: Is-A instead of Has-A
Type test and type guard in Java

- **Type test:** Inquiry of the dynamic type
- **Type guard:** runtime checking of type casting

Example:

```java
if(customer instanceof CorporateCustomer){  // test
    CorporateCustomer corpCust = (CorporateCustomer)customer;  //guard
    ...
}

if(customer instanceof CorporateCustomer)
    ((CorporateCustomer)customer).report(monthlyReport);
```
Understanding Interactions Between Objects
Object Game

Play a hotel room reservation scenario