Inheritance
Software Engineering
CITS1220
Lecture Outline

- **Reuse via inheritance of classes**
- Inheritance in UML and Java
- Polymorphism
Software Reuse

- A main goal of OOP is reuse of software and the concept of software components
- Techniques for software reuse include
  - Cut-n-paste (not recommended!)
  - Aggregation (see previous lecture)
  - Inheritance
Inheritance

- Inheritance is used to build
  - Specializations of existing classes
  - Closely related groups of classes
- Often described as the “is a” relationship
  - Use inheritance when classes A and B satisfy
    - Every A is a B
Accounts

- A bank has a variety of accounts, such as savings, chequing and mortgage accounts.
- Much functionality is similar, but certain types have specialized behaviour.
- Passes the “is a” test in that
  - A MortgageAccount is a Account
  - A SavingsAccount is a Account
Inheritance in UML

Account

SavingsAccount  ChequingAccount  MortgageAccount

Account is called the superclass, while the others are all called subclasses.
Inheritance concepts

- Write the common code (both variables and methods) in the class `Account` describing the properties and behaviours shared by *any type* of account.
- These are automatically inherited by the subclasses, which can optionally add:
  - *Additional* variables and methods, or
  - *Alternative* and/or specialized versions of *existing* methods.
Example

Account

- balance
- openedDate
- creditOrOverdraftLimit

- credit()
- debit()

SavingsAccount

CheckingAccount

- highestCheckNumber
- withdrawUsingCheck()
- calculateServiceCharge()

MortgageAccount

- collateralProperty
- collateralValue

- setCollateralValue()
public class Account {
    private int balance;
    private Date openedDate;

    public void credit(int amount) {
        balance += amount;
    }

    // omitted code
}
In Java

class CheckingAccount extends Account{
    private int highestCheckNumber;

    public void withdrawUsingCheck(int amt,
        int checkNumber) {
        // code omitted
    }
}

- The subclass definition just contains the *additional* variables and methods
Inherited variables

- **Objects of class** `CheckingAccount` **will automatically **inherit** the balance variable
- **Methods in** `CheckingAccount` **can use it**

```java
public void withdrawUsingCheck(int amt, int checkNumber){
    if (amt > balance) {
        bounceCheck();
    }
}
```
Inherited methods

- **Objects of class** CheckingAccount **will automatically inherit** all methods

```java
public void withdrawUsingCheck(int amt, int checkNumber) {
    if (amt > balance) {
        bounceCheck();
    } else {
        debit(amt);
    }
}
```
Inheritance and constructors

- If class B extends class A then construction of a B object involves construction of an A object.
- In fact, the basic A object should be built first and the extra B bits added on afterwards.
- This is accomplished through the use of the keyword `super` which should be the first statement of the constructor of the subclass.
Example

- Suppose that `Account` has a constructor
  \[\text{Account}(\text{int num, int bal})\]
- While a `MortgageAccount` needs an extra value to store the value of the collateral
  \[\text{MortgageAccount}(\text{int num, int bal, int coll})\]
Example cont.

MortgageAccount(int num, int bal, int coll)
{
    super(num,bal);
    collateralValue = coll;
}

- The “super” part runs the constructor of the superclass which sets up all the inherited values; then the rest of the constructor completes the specialized parts
Inheritance hierarchy

MathematicalObject

Shape

Shape2D

Ellipse

Circle

Shape3D

Line

Plane

Point

Matrix

Shape2D

Polygon

Quadrilateral

Rectangle
In Java

- In Java *everything* of reference type is a subclass of the root class `Object`
- The class libraries form an extensive and deep inheritance hierarchy
Liskov Substitution Principle

- A program written using variables of a superclass (e.g. `Account`) should work unchanged regardless of whether the instances are from the superclass or any of its subclasses

  - In other words subclasses should not change the meaning of the inherited fields/methods
In Java

- In Java, a variable of type A can refer to objects of type A or any of its subclasses

```java
Account[] a = new Account[3];
a[0] = new Account(1155, 200);
a[1] = new MortgageAccount(1156, 100, 500000);
a[2] = new CheckingAccount(...);
```
Reusing code

- Good for code reuse
  - Existing code that just uses the methods from `Account` need not be changed even if a new subclass is created
- Write code at the right level of generality
  - Some functions (e.g. mailing out statements to account holders) need not distinguish between the different types of `Account`
Method overriding

- A subclass can *re-define* an existing method as well as add new ones - *overriding* the original

- For **Extension**
  - A `CheckingAccount` has additional government fees on *all* debits and credits, so these methods should be rewritten

- For **Optimization**
  - Deciding whether a point is inside a `Rectangle` is much easier than deciding if it is inside a `Polygon`
Method resolution

Account a = new CheckingAccount(132, 2000);
a.debit(200);

Does this use the plain vanilla “Account” implementation or does it use the specialized “CheckingAccount” version?
Dynamic binding

- Java uses *dynamic binding*
- The choice of implementation is made at run-time depending on the *actual class* of the object, not the variable
- In other words, if the object *IS* a CheckingAccount, then it will use the CheckingAccount implementation
Polymorphism

- This behaviour is called polymorphism
  - This means “many types”
- A single variable can hold different types of object - objects of that class or any of its subclasses
  - When using a polymorphic variable, dynamic binding ensures that the most appropriate method implementation is used