Inheritance

CPSC 1181 – O.O.

Jeremy Hilliker

Summer 2017

Langara.

THE COLLEGE OF HIGHER LEARNING.

Overview

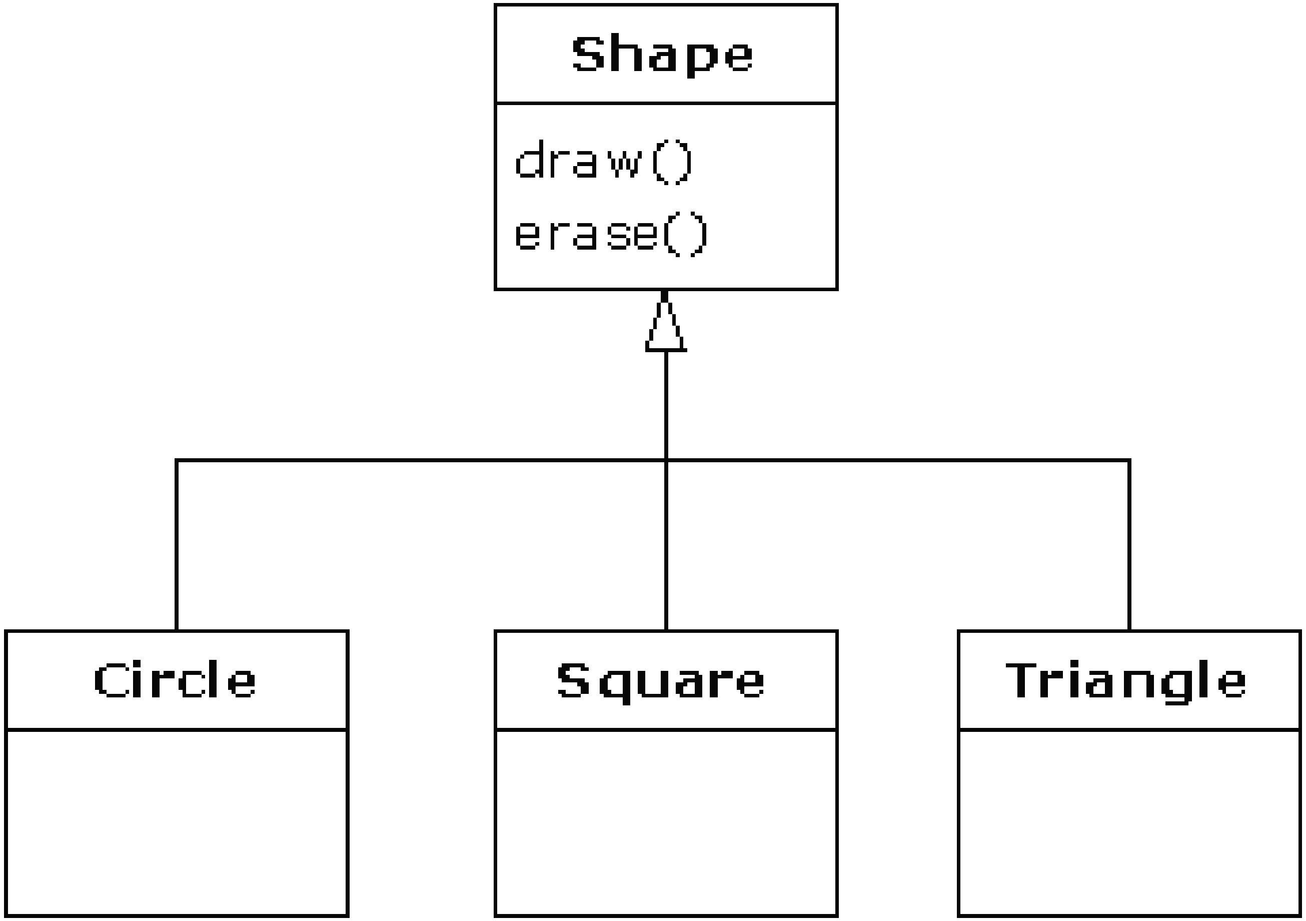
- Motivation
- Specialization & Generalization
- Inheritance
  - Access modifiers
  - Overriding
  - Construction

- Implementation

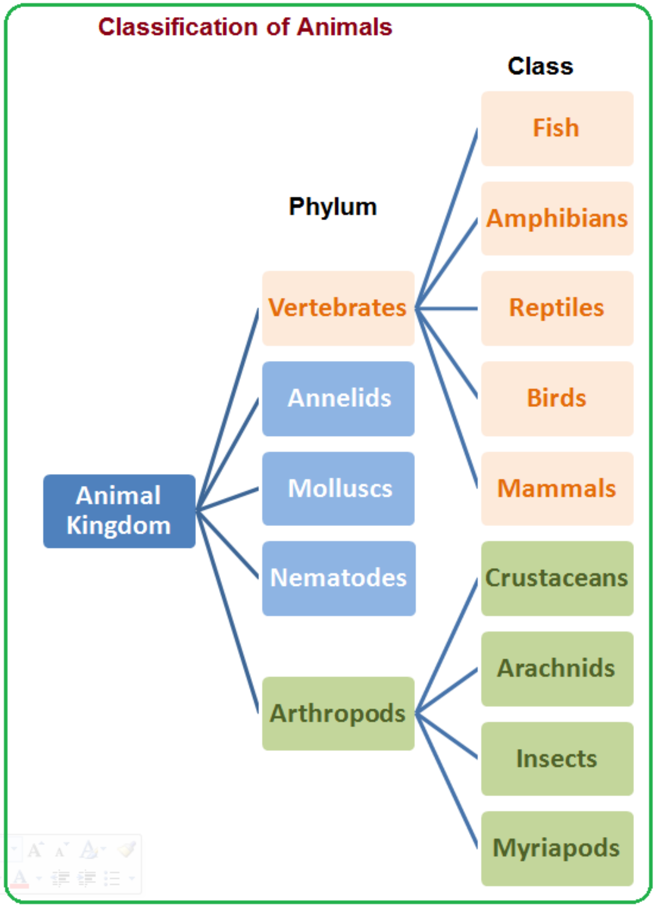
Motivation

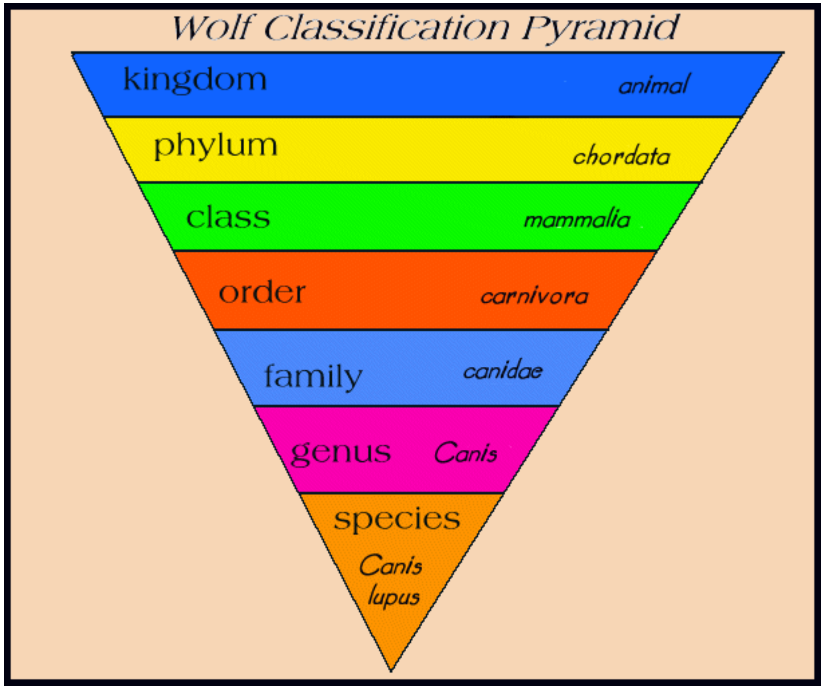
- A Class is a type
- A type is a collection of values and the operations on those values.
- What if we want to take a type and extend it to include more values and operations?
- Or, what if we have a few related types that share some subset of values and operations?
    - Can we reuse them somehow rather than duplicate them?

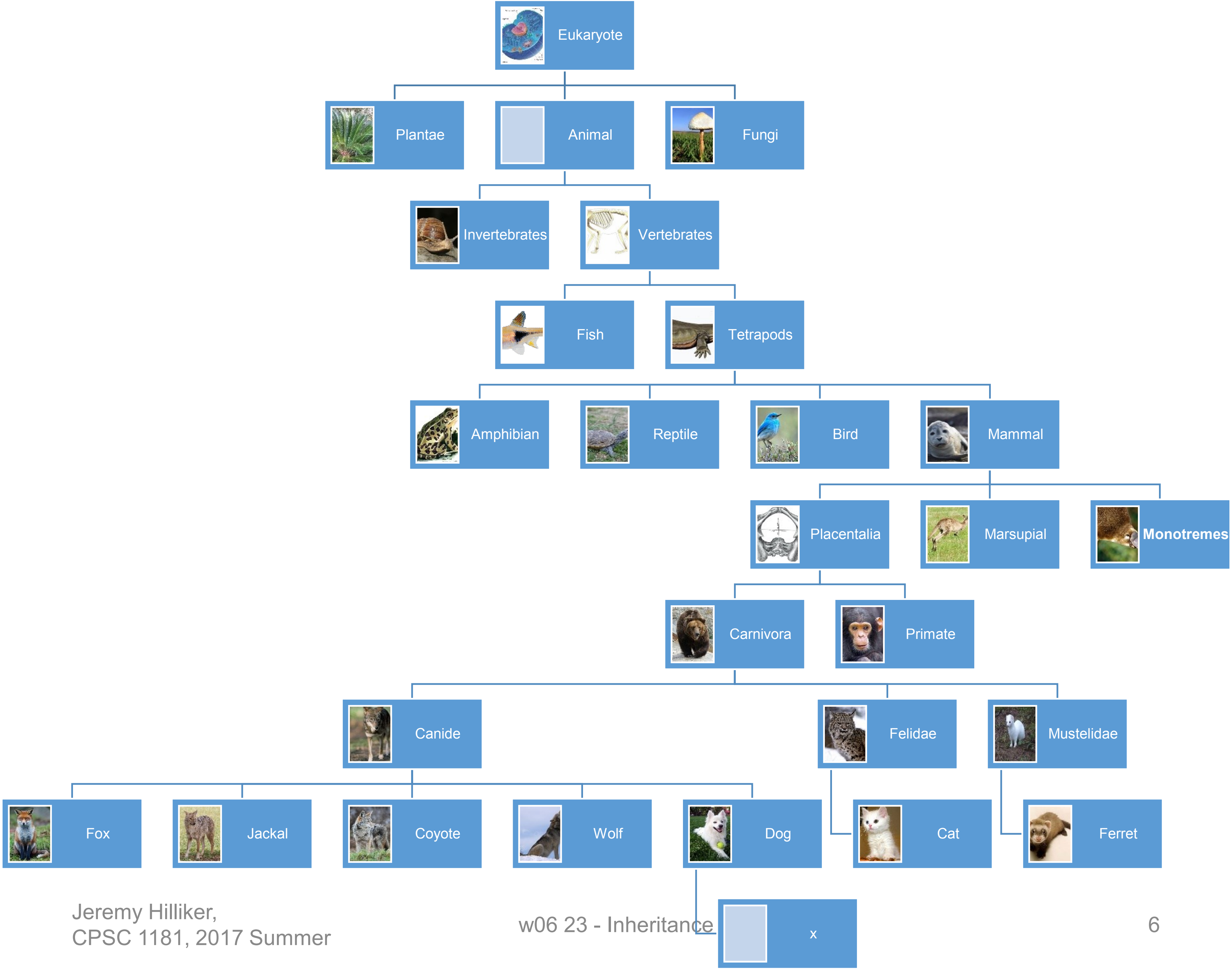
Ex: Shapes



Ex: Animals


Classification of Animals
Class
Phylum
Animal Kingdom
Vertebrates
Annelids
Molluscs
Nematodes
Arthropods
Fish
Amphibians
Reptiles
Birds
Mammals
Crustaceans
Arachnids
Insects
Myriapods


Wolf Classification Pyramid
kingdom
animal
phylum
chordata
class
mammalia
order
carnivora
family
canidae
genus
Canis
species
Canis lupus



- Eukaryote
  - Plantae
  - Animal
    - Invertebrates
    - Vertebrates
      - Fish
      - Tetrapods
        - Amphibian
        - Reptile
        - Bird
        - Mammal
          - Placentalia
            - Carnivora
              - Canide
                - Fox
                - Jackal
                - Coyote
                - Wolf
                - Dog
                  - x
              - Felidae
                - Cat
              - Mustelidae
                - Ferret
            - Primate
          - Marsupial
          - **Monotremes**
  - Fungi

Specialization & Generalization

- Going up the tree can be described as:
    - **Is a** *type* of
        - Sedan is-a Car, Car is-a Vehicle
        - Mammal is-a Vertebrate, Vertebrate is-a Animal, Animal is-a Eukaryote
- Going down the tree is seen as specialization
    - Sedan is-a specialization of Car
    - Mammal is-a specialization of Vertebrate
- Going up is, therefore, generalization
    - Vehicle is more generalized than car

Programming Nomenclature

- We call the more specialized type:
  - A ***subtype***
    - Car is-a subtype of Vehicle
- We call the more generalized type:
  - A s***uper-type***
    - Vehicle is a* super-type of Car
- Likewise for class:
  - ***subclass***, ***super-class***

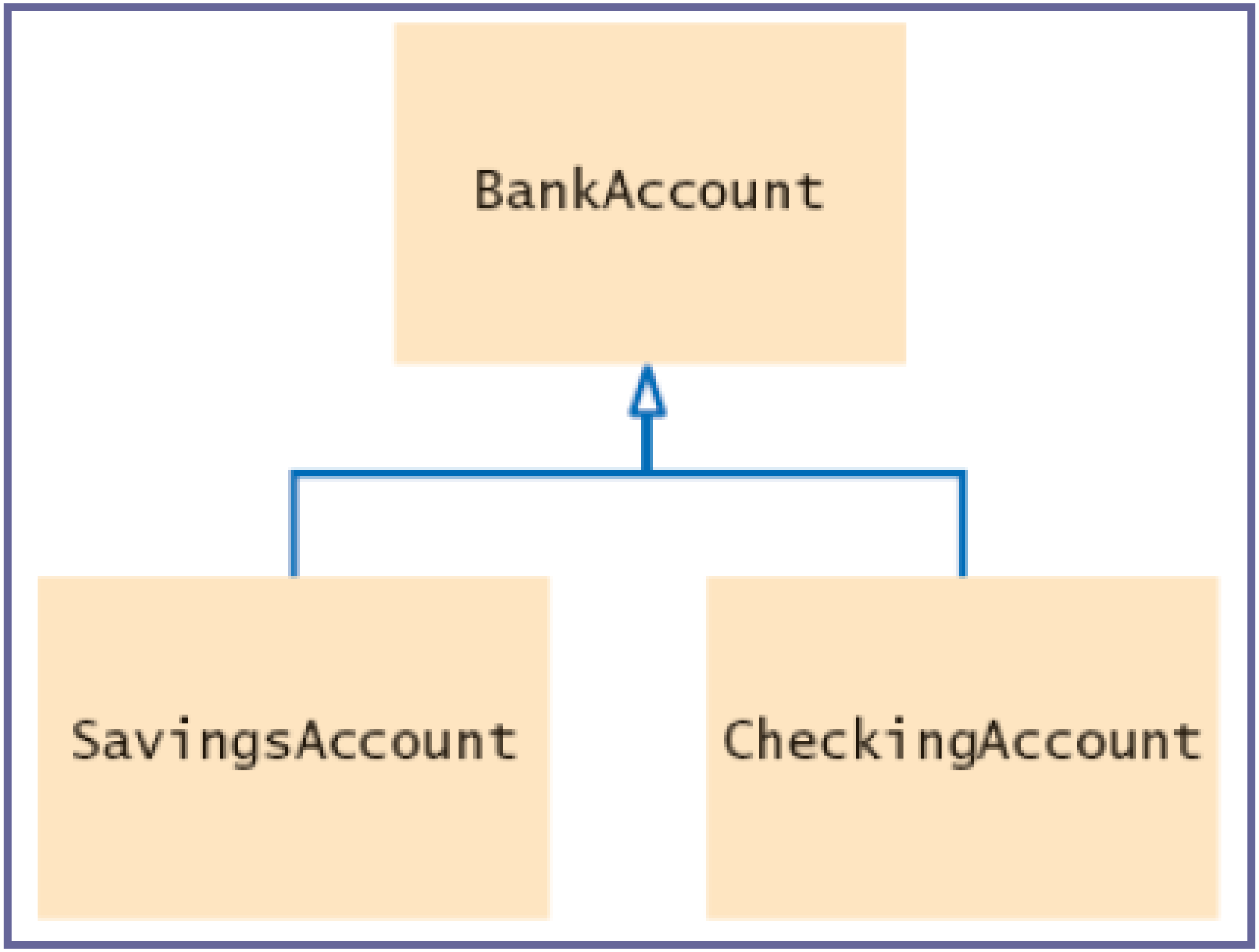
Programming Nomenclature

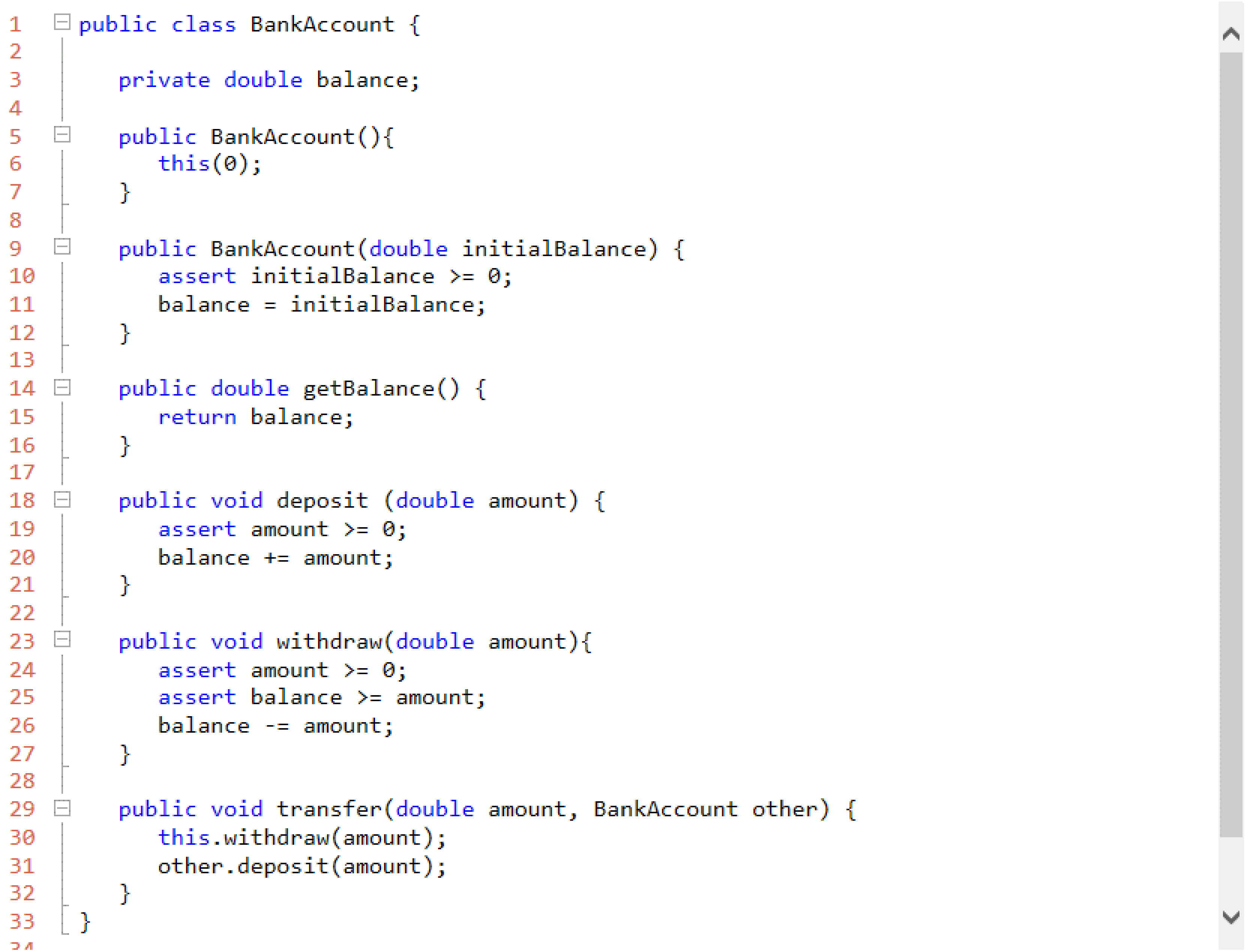
- A subtype ***inherits*** its super-type's values and operations
  - Car shares the common traits of Vehicle
  - Mammal shares the common traits of Vertebrates, Animals, and Eukaryotes.
- A subtype ***extends*** those values and operations by adding its own
  - Not all animals purr, but cats do
- In general, we would use "***is-a"*** and "***extends"*** to describe the relationship
  - Car is-a Vehicle; Car extends Vehicle
  - BUT NOT: Vehicle is-a Car

Note

- In java,
  - Every object is-a Object
  - Every class extends from Object
    - Either directly or indirectly.
  - Objects is a* super-type to all object types in java

Consider





```
public class BankAccount {

    private double balance;

    public BankAccount(){
        this(0);
    }

    public BankAccount(double initialBalance) {
        assert initialBalance >= 0;
        balance = initialBalance;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit (double amount) {
        assert amount >= 0;
        balance += amount;
    }

    public void withdraw(double amount){
        assert amount >= 0;
        assert balance >= amount;
        balance -= amount;
    }

    public void transfer(double amount, BankAccount other) {
        this.withdraw(amount);
        other.deposit(amount);
    }
}
```

Ex: SavingsAccount

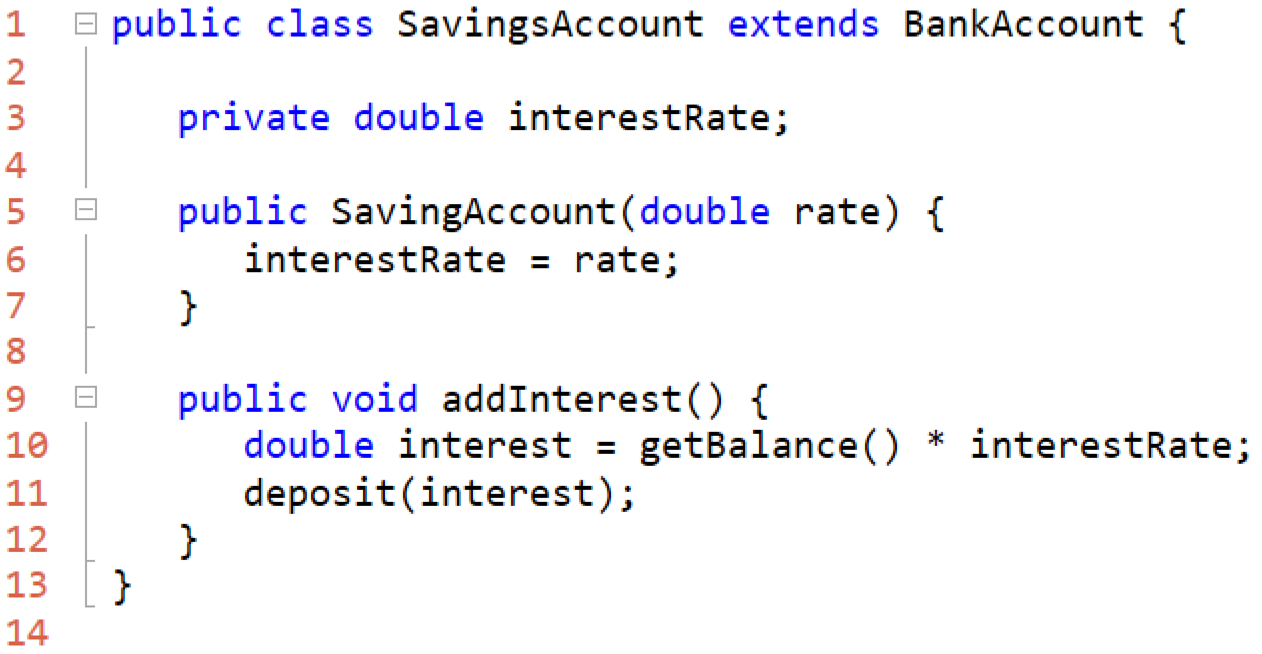
BankAccount

- Attributes:
  - balance
- Behaviours:
  - deposit
  - withdraw
  - transfer

SavingsAccount

- Attributes:
  - interest rate
- Behaviours
  - apply interest

Ex: SavingsAccount

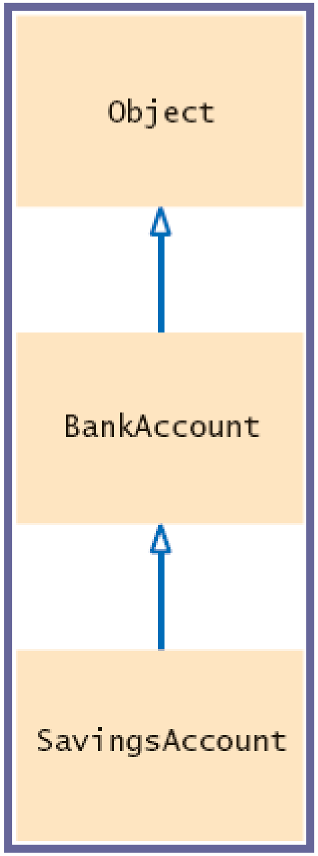


```java
public class SavingsAccount extends BankAccount {

   private double interestRate;

   public SavingAccount(double rate) {
      interestRate = rate;
   }

   public void addInterest() {
      double interest = getBalance() * interestRate;
      deposit(interest);
   }
}
```

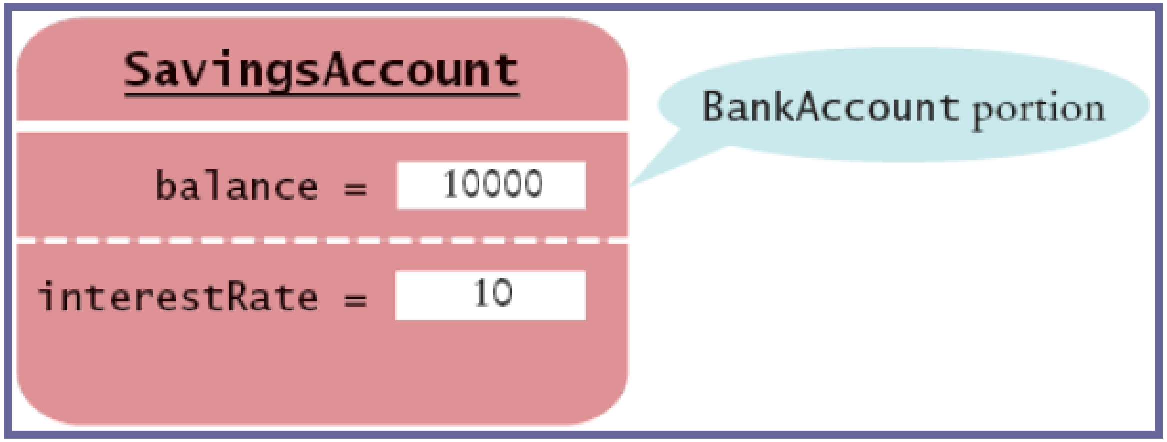


Inheritance

- SavingsAccount *inherits* the values* and behaviours* of its super-type, BankAccount

  - balance*

  - deposit
  - withdraw
  - transfer



Inheritence

- Code reuse:
  - SavingsAccount does not have to duplicate the code in BankAccount
    - It *inherited* the behaviours from BankAccount
- Cohesion:
  - Other types of BankAccount (like ChequingAccount) aren't exposed to SavingsAccount's specializations
    - interestRate, addInterest()
  - All of SavingsAccount's behaviours are related to its specializations
    - interestRate, addInterest()

Encapsulation

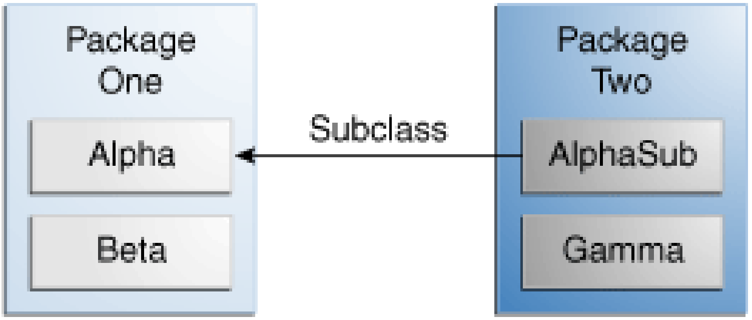
- SavingsAccount *could not* directly update the “balance” instance variable
  - “balance” belongs to BankAccount
  - BankAccount has declared that only it may modify the “balance” instance variable
    - private
  - This is generally a good idea
- SavingsAccount could access getBalance() and deposit()
  - because BankAccount declared that others could access those behaviours.

Access Modifiers

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	
<no modifier>	✓	✓		
private	✓			

- <no modifier> is called “package-private”
- Stick to public and private
  - unless you have a very good reason
    - (read: your reason is probably bad).

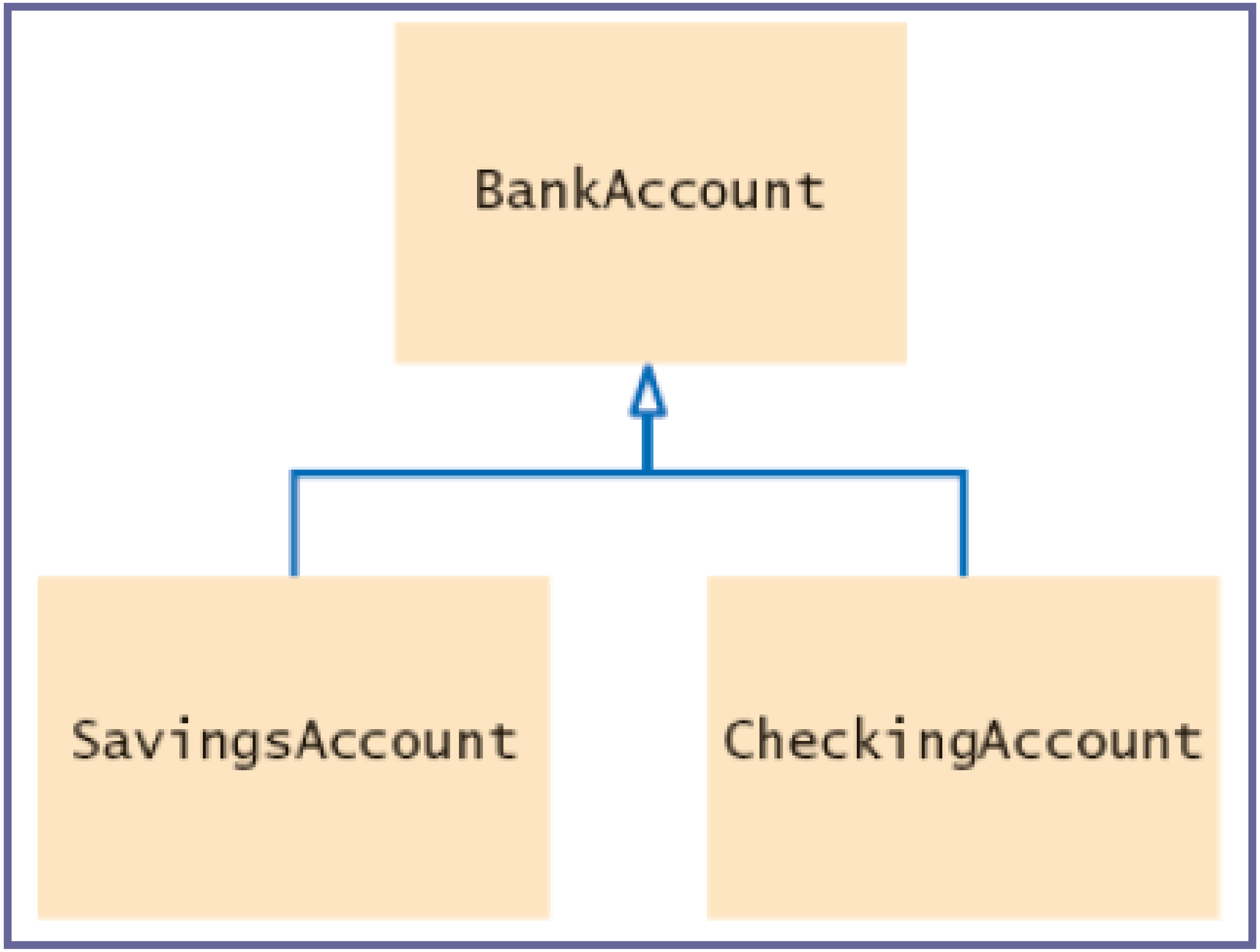
Ex: Access Modifiers



https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Modifier	Alpha	Beta	AlphaSub	Gamma
public	✓	✓	✓	✓
protected	✓	✓	✓	
<no modifier>	✓	✓		
private	✓			

Consider



Ex: CheckingAccount

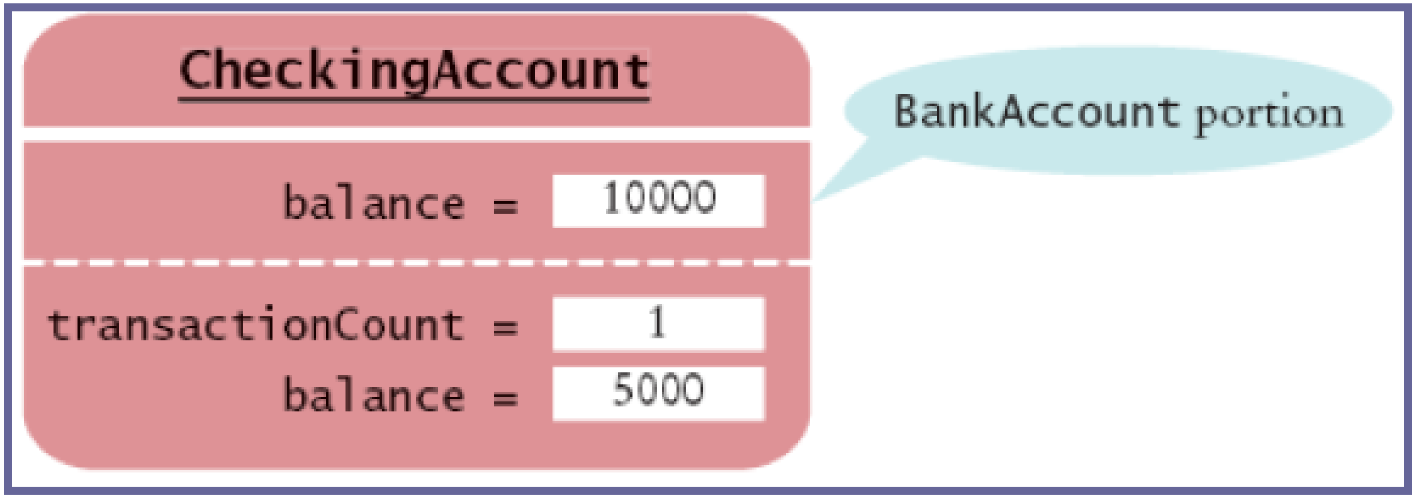
- BankAccount
  - Has-a: balance
  - Can: deposit(), withdraw(), transfer()
- SavingsAccount (w/r/t BankAccount)
  - Adds has-a: interestRate
  - Adds can: applyInterest()
- CheckingAccount (w/r/t BankAccount)
  - Adds has-a: transactionCount
  - Adds can: deductFees()
  - Changes can: deposit(), withdraw(), transfer()

Overriding

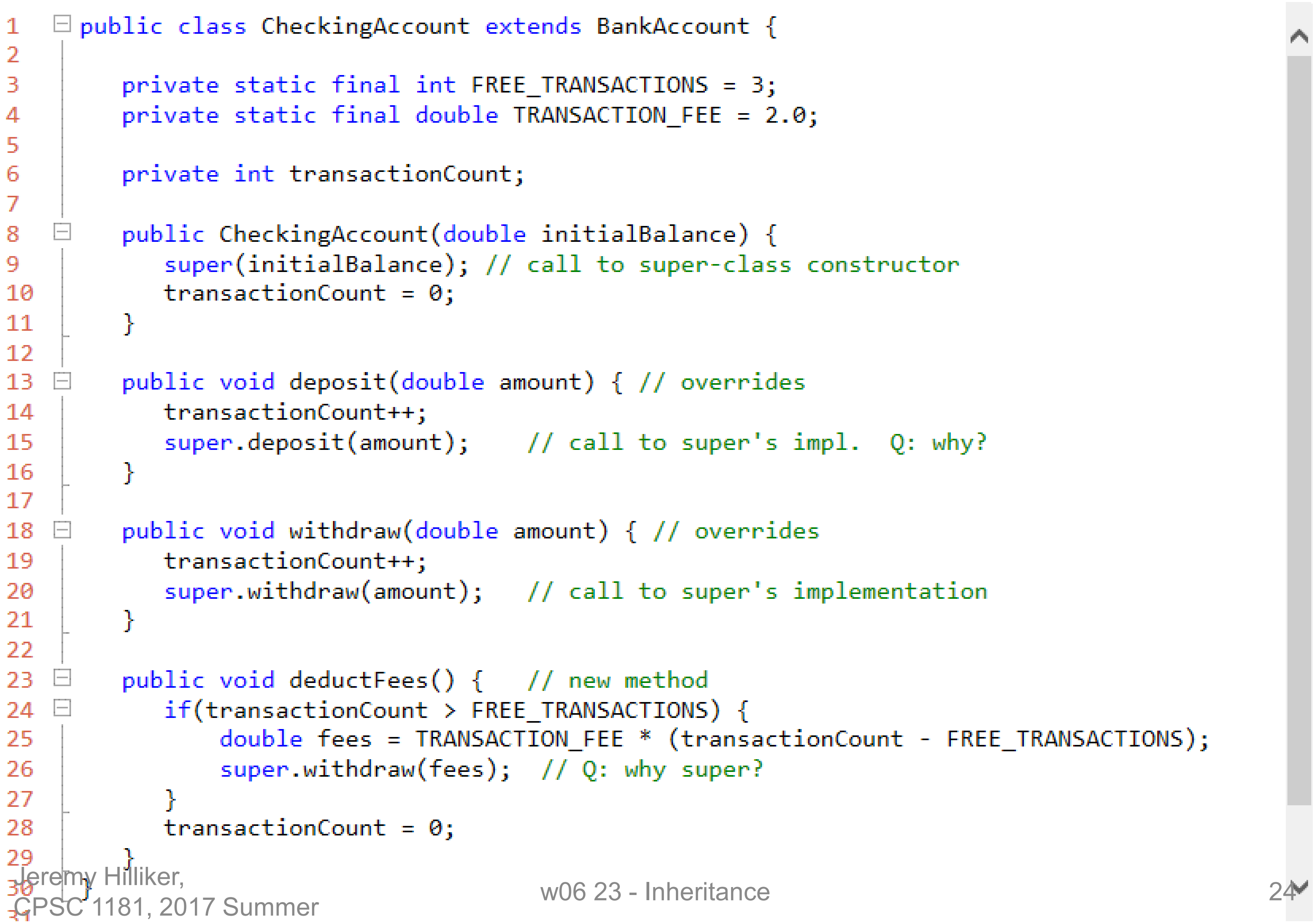
- A subclass can ***override*** the behaviour specified by the super-type
- The subclass does this by defining its own implementation of the method (with the same signature)
- The subclass can still access the superclass' implementation by referring to "super" (as opposed to "this")
  - super.equals(…)
- Applies to methods only

Cannot Override Fields

- Trying to override a field is called variable shadowing.
  - It’s generally an error (legal, but a really bad idea)
  - Follows normal scope rules
    - If you declare your own, then you use yours
    - If you don’t declare your own, you use the inherited



Ex: CheckingAccount



```java
public class CheckingAccount extends BankAccount {

    private static final int FREE_TRANSACTIONS = 3;
    private static final double TRANSACTION_FEE = 2.0;

    private int transactionCount;

    public CheckingAccount(double initialBalance) {
        super(initialBalance); // call to super-class constructor
        transactionCount = 0;
    }

    public void deposit(double amount) { // overrides
        transactionCount++;
        super.deposit(amount);    // call to super's impl.  Q: why?
    }

    public void withdraw(double amount) { // overrides
        transactionCount++;
        super.withdraw(amount);   // call to super's implementation
    }

    public void deductFees() {   // new method
        if(transactionCount > FREE_TRANSACTIONS) {
            double fees = TRANSACTION_FEE * (transactionCount - FREE_TRANSACTIONS);
            super.withdraw(fees);  // Q: why super?
        }
        transactionCount = 0;
    }
}
```

Constructors

- The super-type must be initialized first in case the subtype relies on it
- So, must call another constructor on the first line
  - this(…)
  - super(…)
- If not, java inserts a call to the super-class' *default constructor*
  - If it doesn't exist, it's a compile error in the subclass
- ***Default constructor***
  - Constructor with no parameters

Recap

- Motivation
- Specialization & Generalization
- Nomenclature
- Inheritance
  - Encapsulation
    - Access modifiers
  - Overriding
  - Constructors
- Ex Implementation:
  - BankAccount
  - SavingsAccount
  - CheckingAccount