

Assertions and Inheritance: Overview

- **Assertions**
 - * **Programming by Contract**
 - * **Assertions on Methods**
 - * **Assertions Inside Methods**
 - * **Assertions on Classes**
 - * **Exceptions**
- **Inheritance**
 - * **Static vs. Binding**
 - * **Inheritance in Eiffel**
 - * **Assertions and Inheritance**

Programming by Contract

- **Contract governs interactions between parties.**
 - * **Defines legal interactions between participants.**
- **Contracts document rules for an operation or interaction.**
 - * **Define conditions necessary for correct use.**
- **Assertion: Logical expression that describe actions of software.**
 - * **Ideally, boolean conditions tested by environment.**
 - * **In Java, sometimes must be stated as comments.**
- **Precondition: Assertion that must be true whenever a method is called.**
 - * **typically a Javadoc comment**

- * **Conditions for correct operation of method.**
- **Postcondition: Assertion that must be true when the method has completed operation.**
 - * **Ideally, boolean conditions tested by environment.**
 - * **In Java, implied or stated in comments.**
 - * **Properties in the object that hold when the method has completed execution.**
- **Pre and postconditions define a contract that binds the method and its callers.**
 - * **With respect to the state of an object and the actions of the method.**
 - * **Clear delineation of responsibility of the method.**
 - * **Up to the caller to ensure the method is called correctly.**

- **If a precondition is not satisfied, the method is not bound to fulfill the contract.**
 - * **Does not have to fulfill the postcondition.**
- ```

/** adds amt to balance
@param amt amount to add
@throws IllegalArgumentException if amt <= 0*/
public void deposit(double amt) throws IllegalArgumentException {
 //necessary to store value for later use
 double origBalance = balance;
 //precondition check of public method
 if (amt <= 0) {
 throw new IllegalArgumentException("amount " + amt
 + " must be greater than zero");
 }
 balance += amt;
 //postcondition
 assert balance == origBalance + amt : "Addition error";
}

```

- **Assertions are evaluated like any other boolean expression.**
  - \* **Evaluations can be enabled via command line argument.**
- **If assertion or if statement evaluates to false, the contract has been violated.**
  - \* **If assertion violated, *AssertionError* is thrown.**
  - \* **Interrupts program execution and transfers control to exception handling method.**
  - \* **Default response is program termination with message describing location of violation.**

Exception in thread "main" java.lang.IllegalArgumentException: amount -100.0 must be greater than zero

```
at Account.deposit(Account.java:12)
at Account.main(Account.java:29)
```

## Programming by Contract

- **In programming by contract:**
  1. **The client and supplier enter into a mutually beneficial contract.**
  2. **The client agrees to only call a supplier's feature when the precondition is true.**
    - \* **The supplier then agrees to guarantee the postcondition.**
  3. **The client gets the supplier to do some work for her.**
    - \* **e.g. put an element into an array at a certain index.**
  4. **The supplier gets to assume the precondition and hence write simpler code. Supplier often isn't in a position to determine the proper response to an invalid condition.**
    - \* **e.g. The supplier doesn't have to check the index valid.**

## Assertions on Methods

- **Java uses parameters and Javadoc comments to write contracts for the behavior of methods.**
- **Note the postcondition allows the caller to understand what the method does without inspecting the implementation.**

*Javadoc output -- viewed in web browser*

```

deposit

public void deposit(double amt)
 throws java.lang.IllegalArgumentException

 adds amt to balance

Parameters:
 amt - amount to add
Throws:
 java.lang.IllegalArgumentException - if amt <= 0

```

## Assertions on Methods

- **Monitoring assertions is intended to catch unintentional violations.**
- **Callers must be able to evaluate preconditions.**
  - \* **Preconditions should only reference features that are available to callers (public interface).**
  - \* **Phrase preconditions in terms of methods/attributes in public interface.**
  - \* **Don't include implementation details in preconditions.**
  - \* **Ask what must be true about the state of the object and any parameters before calling a method.**

## Assertions on Methods

- **Postconditions provide two things:**
  - 1. Enumerate the method's obligations to its callers.**
    - \* **Expressed in terms of the public interface.**
  - 2. Provide a check on the implementor's assumptions.**
    - \* **Record changes not visible to caller.**
    - \* **May contain implementation features.**
- **Ask what changes about the state of the object after the execution of the method.**

## Assertions on Methods

- **Labeling assertions provides information when violations occur.**

```
assert balance == origBalance + amt : "Addition error";
produces error
```

```
Exception in thread "main" java.lang.AssertionError: Addition error
at Account.deposit(Account.java:17)
at Account.main(Account.java:27)
```

- **Assertion can be written without label**

```
assert balance == origBalance + amt ;
but it provides less information when error occurs.
```
- **Postconditions should describe all legal changes.**

## Assertions Inside Methods

- **We can gather information about the internal state of a method via the assert keyword.**
- **may be used anywhere inside a method.**
- **Documents programmer's assumptions about state of object at that point.**
- **As with all assertions, must enabled via command line argument.**

## Assertions Inside Methods

- **If monitoring is on, the assert instruction will be executed.**
  - \* **If assertion is true, execution proceeds.**
  - \* **If assertion is false, exception is raised.**
- **Common example is divide by zero.**

**Programmer knows that denominator might be zero:**

```
if (x !=0) {
 y = z/x;
} else {
 //error handling here
}
```

**If the denominator should never be zero:**

```
assert x!=0;
```

## Assertions on Classes

- **In addition to describing the requirements and effects of a method, Java allows supports of assertions on classes.**
- **Class Invariants: Assertions that describe the properties that must hold stable at all times.**
  - \* **Properties that must be true at beginning and end of each interaction between class and client.**
  - \* **Do not have to hold during execution of a method.**
  - \* **Describe the “general principles of operation” of the class.**
- **Ask “What are the normal, consistent states of the object?”**

## Java Class Invariants

- **Recommended strategy is to write boolean method to describe the proper state(s) of the object.**

```
private boolean validState() {
 ...}

```

- **Then assert at beginning and end of methods.**

```
public void deposit(double amt) throws IllegalArgumentException {
 assert validState();
 //method code
 assert balance == origBalance + amt : "Addition error";
 assert validState();
}

```

## Assertion Monitoring

- **Assertions only available with latest release of Java (1.4)**
- **In order to compile code with *assert* keyword, must use the *source* switch**

```
javac -source 1.4 Account.java
```

- **In order to active runtime monitoring, must use the *enableassertions* switch -- which may be abbreviated *ea***

```
java -ea Account
```

- **In BlueJay, assertions are enabled from the Tools->Preferences menu item.**

## Exceptions: Overview

- **Purpose of Exceptions**
- **The Throw/Catch Model**
- **User-Defined Exceptions**
- **The Error Class**
- **The finally Clause**
- **When to use Exceptions**
- **Polymorphism: Create a user-defined exception.**

## Purpose of Exceptions

- **There are three problems with using return codes for error handling:**
  1. **Complexity: Caller must check the return code and must include logic to handle each different return code.**
    - \* **Every procedure in chain of calls has to deal with the error.**
  2. **Limited Information: An error code can only include so much information.**
    - \* **Error code becomes more out of context the further up the calling chain it is passed.**
  3. **Used up returns: Forces actual return value to be passed back some other way.**

## Purpose of Exceptions

- **Java provides an exception handling approach for detecting and reporting errors.**
- **Exception: Error condition that interrupts flow of program.**
- **Java creates a special Exception object when something goes wrong in a program.**
- **Exceptions are used for changing the flow of control when an unexpected event occurs.**
- **Exceptions divert processing to a part of the program that can try to cope with the error.**
  - \* **Or at least die gracefully.**
- **Error conditions could be:**
  - \* **unable to open a file**

- \* array subscript out of range**
- \* no memory left to allocate**
- \* divide by zero**

## The Throw/Catch Model

- **Java uses the throw/catch model of exception handling.**
  - \* Some other languages call it the raise/handle model.**
- **Exceptions are caused in one of two ways:**
  - 1. Program does something illegal (usual case).**
  - 2. Program executes the throw keyword.**

```
public class DivideByZero
{
 public static void main (String[] args) {
 int i = 1;
 int j = 0;
 int k;
 k = i/j; // Forcing divide by zero error
 }
}
```

```
}

```

```
Exception in thread "main" java.lang.ArithmeticException: / by
zero
at DivideByZero.main(DivideByZero.java:14)
•

```

- **Java provides a number of predefined exceptions known as runtime exceptions.**

\* e.g., **Arithmetic Exception**

- **In order for exception handling to occur, a try/catch block must be used.**

```
try {
 // Code that may produce an exception
}
catch (Exception e) {
 // Code that handles the problem
}

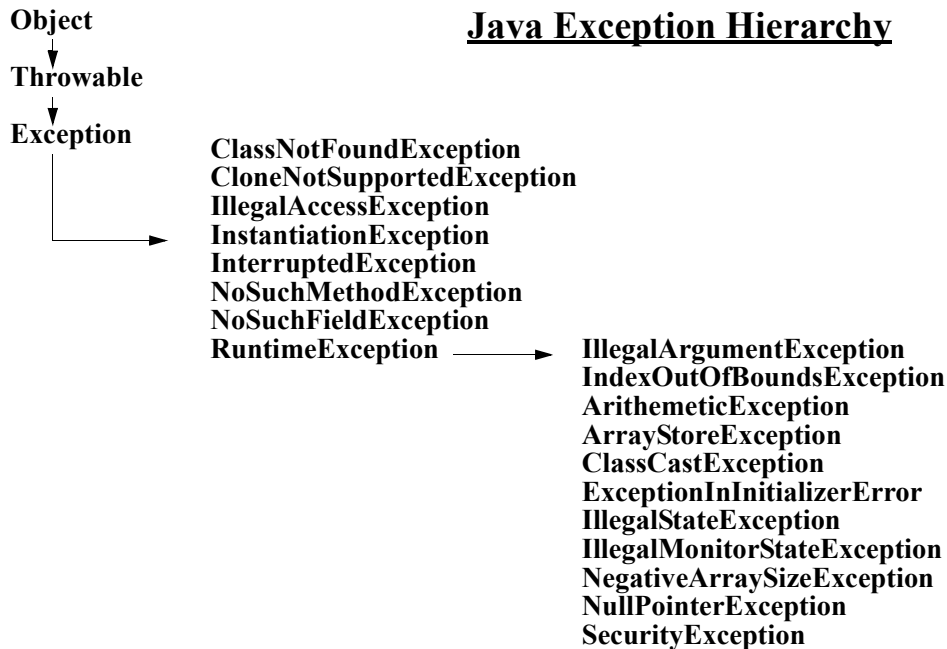
```

- **The try statement says “try these statements and see if you get an exception.”**
- **try block encloses a series of statements from which a throw can originate.**
- **The try statement must be followed by at least one catch or finally clause.**
  - \* **catch block must immediately follow a try block or another catch block.**
- **catch keyword says “I will handle any exception that matches my argument.”**
  - \* **Section of code designed to process a thrown exception.**
- **When an exception is thrown via the throw statement within a try block, control is passed to the associated catch block.**

- \* Runtime exceptions not caught unless explicitly thrown.**
- You may have several successive catches, each catching a different exception.**
- If there is no catch block in current method, control exits immediately to calling method without returning a value.**
- Control passes up through calling stack until Java finds a catch block capable of handling the error.**
- Advantages to throw/catch model:**
  - 1. Error conditions dealt with only where it makes sense to deal with it.**
    - \* Don't have to deal with error at every level between occurrence and where it is handled.**

- 2. Code can be written as if the operations in it will work.**
  - \* Separates error handling from normal flow of control.**
- 3. Reduces program complexity.**
  - \* Calling functions do not need to error check returned values.**

# User-Defined Exceptions



- **Argument to throw can be any expression that returns an instance of Throwable or its subclasses.**
  - \* **Typically Exception class or subclass.**
- **Throwable class provides:**
  - \* **A slot for a message: Should describe the error.**
  - \* **A stack trace.**
- **You can create your own exception subclasses for handling special cases.**
  - \* **Typically create a subclass of Exception.**

- **The PositiveWithdrawalException:**

```

import java.io.*;
public class PositiveWithdrawalException extends Exception {
 protected Account myAccount;
 protected double withdrawalAmount;
 public PositiveWithdrawalException(Account acct, double amt)
 {
 super("Withdrawal amount must be positive.");
 myAccount = acct;
 withdrawalAmount = amt;
 }
 public String toString()
 {
 StringBuffer sb = new StringBuffer();
 sb.append("\nAccount number: ");
 sb.append(myAccount.getAccountNumber());
 sb.append("\nBalance was: ");
 sb.append(myAccount.getBalance());
 sb.append("\n Withdrawal was: ");
 sb.append(withdrawalAmount);
 return sb.toString();
 } }

```

- **First we'll look at the case where the exception is both thrown and caught locally:**

```

public class Account {
 . . .

 // Method to allow a positive amount to be withdrawn from the account.
 public void withdraw(double amt)
 {
 try {
 if (amt < 0.0)
 { throw new PositiveWithdrawalException(this, amt); }
 balance -= amt;
 }
 catch(PositiveWithdrawalException pwe) {
 System.err.println("Error : " + pwe.getMessage());
 pwe.printStackTrace();
 }
 }
}

```

```
public class Setup
{
 public static void main(String[] arg)
 { Account theAccount = new Account(123);
 theAccount.deposit(200);
 theAccount.withdraw(-300);
 } // End the main function.
}
```

```

laplace% java Setup
Error : Withdrawal amount must be positive.
Account number was: 123
Balance was: 200.0
Withdrawal was: -300.0
 at java.lang.Throwable.fillInStackTrace(Native Method)
 at java.lang.Throwable.<init>(Throwable.java:94)
 at java.lang.Exception.<init>(Exception.java:42)
 at PositiveWithdrawalException.<init>(PositiveWithdrawalException.java:17)
 at Account.withdraw(Account.java:41)
 at Setup.main(Setup.java:19)
```

- **Now lets look at the situation where the exception is raised within a method and is thrown to the calling method:**

```
public class Account {
 . . .

 // Method to allow a positive amount to be withdrawn
 // from the account.
 public void withdraw(double amt) throws PositiveWithdrawalException
 {
 if (amt < 0.0)
 {
 throw new PositiveWithdrawalException(this, amt);
 }
 balance -= amt;
 }

 . . .
}
```

```

public class Setup
{
 // Define a public static main function to be the "main" function.

 public static void main(String[] arg)
 {
 // Create an account with an initial balance.
 Account theAccount = new Account(200);

 // Use a try block to attempt to withdraw from the account.
 try {
 theAccount.withdraw(300);
 }
 catch (PositiveWithdrawalException pwe){
 System.err.println("Error : " + pwe.getMessage());
 pwe.printStackTrace();
 }
 } // End the main function.
}

```

```

laplace% java Setup
Error : Withdrawal amount must be positive.

Account number was: 200
Balance was: 0.0
Withdrawal was: -300.0
at java.lang.Throwable.fillInStackTrace(Native Method)
at java.lang.Throwable.<init>(Throwable.java:94)
at java.lang.Exception.<init>(Exception.java:42)
at PositiveWithdrawalException.<init>(PositiveWithdrawalException.java:17)
at Account.withdraw(Account.java:40)
at Setup.main(Setup.java:19)

```

## User-Defined Exceptions

- **In Java, the exceptions a method can throw are part of the public interface.**
- **Java method definitions must include list of exceptions that the method throws.**

```
public void withdraw(double amt)
 throws PositiveWithdrawalException
{ . . . }
```

- **If your method calls another method that can throw an exception, your method must:**

**1. Declare itself capable of throwing the same exception as the called method.**

**- Passes the exception up the calling stack.**

**OR**

**2. Include a try/catch block to make sure the exception does not pass through to caller.**

- **Runtime exceptions: Common exceptions that can occur anywhere in a program.**
  - \* NullPointerException, ArrayIndexOutOfBoundsException
- **You do not need to declare runtime exceptions in method unless you explicitly throw them.**

## The Error Class

- **The Error class is a subtype of Throwable reserved for Java's use.**
  - \* **Used for catastrophic, unrecoverable errors.**
  - \* **Application code should NOT throw Errors.**
- **You can catch errors:**

```
catch (Error e) {
 // Do something here
}
```

## The finally Clause

- **Exceptions can cause control to leave the current method without completing the method's execution.**
- **Problem: You may need to clean up.**
  - \* **e.g., close files, etc.**
- **The finally statement ensures that clean up occurs.**

```
try { // Normal processing here
} catch (Exception e) {
 // Handle errors
}
finally
{ // Executed before control exits and cleans up.
}
```

## The finally Clause

- **The finally statement must be used in conjunction with a try block.**
- **The finally block contains code guaranteed to run if an error occurs.**
- **The finally block is executed before control exits the method even if the try block contains a return statement.**
- **If a break statement is executed within a try block, the finally clause is executed before control is passed.**

```
try{
 . . .
 break;
 . . .
}
finally { . . . }
```

## When to use Exceptions

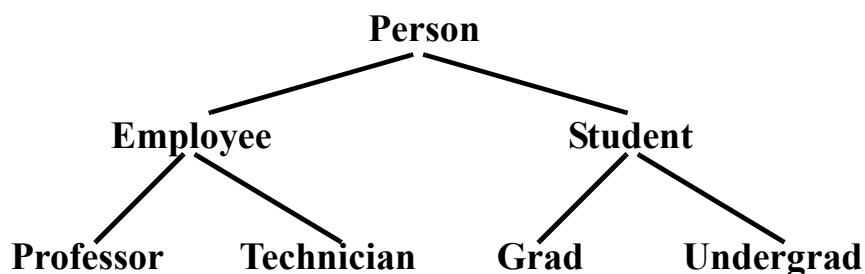
- **Exceptions provide safer program execution by supplying a clear way of handling errors.**
- **Exceptions should be used for organizing error handling.**
- **The main use of exceptions is for describing what, where, and why of errors.**
- **Difficult for exceptions to take care of full recovery.**

# Inheritance: Overview

- **Single Inheritance**
- **Polymorphism**
- **Abstract Classes**
- **Account Example**
- **Final Classes**
- **Determining Type**

## Single Inheritance

- **Frequently when we design, we create two or more classes that have much in common.**
- **We want to reuse the common parts by combining the common parts into one class and reusing that class.**
- **Inheritance: Defining a class that extends the structure and behavior of another class.**



# Single Inheritance

- **Child class or subclass: Class that inherits.**
- **Parent class or superclass: Class from which subclass inherits.**
- **Inheritance hierarchy or graph: Tree of related inheritance relationships.**

|                  |                                                                  |                             |
|------------------|------------------------------------------------------------------|-----------------------------|
| <b>Person</b>    | <b>get_name    set_name<br/>get_age     set_age</b>              | <b>name<br/>age</b>         |
| <b>Employee</b>  | <b>&lt; all from Person &gt;<br/>give_raise    get_salary</b>    | <b>salary</b>               |
| <b>Professor</b> | <b>&lt; all from Employee &gt;<br/>add_advisee    add_course</b> | <b>courses<br/>advisees</b> |

- **New way to reuse a class.**
  - \* Existing class is extended with no change to its own code.
  - \* Existing users of parent class are not affected.
  - \* Child class treats all inherited features exactly as if written in child class.
- **Inheritance is transitive: Class can inherit features from superclass many levels away.**
- **Inheritance implies some relationship of functionality between two classes.**
  - \* The *is-a* relationship is fundamental to the use of inheritance.

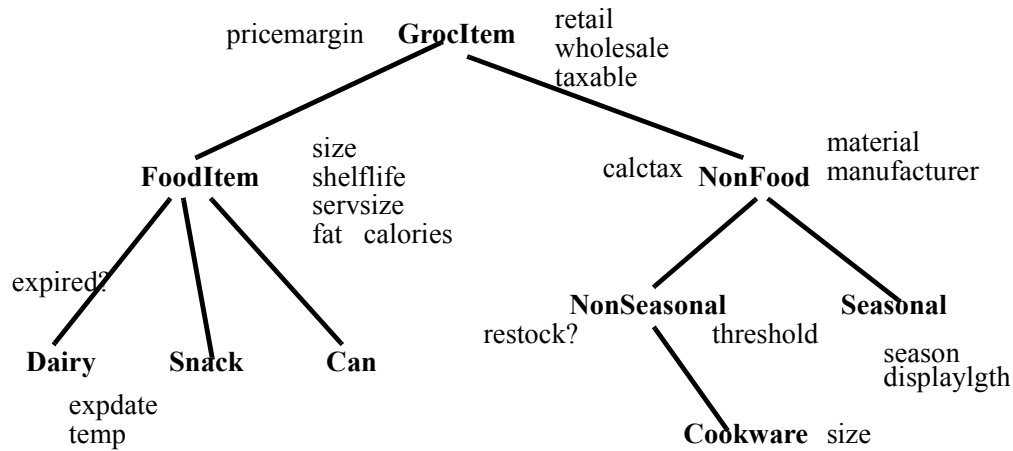
## Single Inheritance

- **Generalization**: The identification and combination of commonalities among similar classes into a common parent class.
  - \* Goes up the hierarchy.
- **Specialization**: The refinement and detailing of a superclass into a subclass.
  - \* Goes down the hierarchy.
- Behavior of child class is always an extension of properties in parent class.
  - \* Larger set of abilities.
- Child class is also a refinement of superclass.
  - \* Customized set of abilities.

## Single Inheritance

- If “A has a B” makes more sense, then objects of B are attributes on A.
  - \* Containment.
- If “A is a kind of B” makes sense, use inheritance.
- When do we use inheritance from an existing class?
  - \* If class used by other applications, subclass.
  - \* If change is transparent, modify existing class.
- Designers working on the same project need to coordinate when using inheritance.
  - \* Overuse of inheritance results in the OO equivalent of spaghetti code.

# Quiz



- **What attributes does Cookware contain? Methods?**
- **What methods can we call on NonSeasonal? Dairy?**

```

// File CheckingAccount.java
// Author: Heidi Ellis
// Creation date: 7/6/99
// Supertype: Account
// Purpose: Represents a checking account that is a specialized type of Account.
// Checking accounts have a minimum balance of $500.00

```

```

public class CheckingAccount extends Account {

// Attributes to hold the minimum balance.
protected static double minimumBalance = 500.00;

// Constructor that takes an account number and initial balance, calls
// superclass constructor and checks initial balance.
public CheckingAccount(int actno, double bal)
{
 super(actno, bal);
 if (bal < minimumBalance)
 { System.out.println("Error: Insufficient initial balance."); }
}
}

```

```

public class Bank
{
 . . .
 public void newCheckingAccount(double amt)
 { // Create a new checking account.
 CheckingAccount myChecking = new CheckingAccount(noAccounts, amt);

 // Add the account to the Bank's collection
 theAccounts[noAccounts] = myChecking;
 noAccounts++;
 }
 . . .
}

public class Setup
{
 . . .
 // Tell the bank to create several accounts with various initial balances.
 theBank.newCheckingAccount(2000);
 theBank.newCheckingAccount(0);
 . . .
}

```

- **Java supports inheritance via the extends keyword.**
- **Multiple inheritance is NOT directly supported.**
- **CheckingAccount inherits all attributes and methods from Account.**
  - \* **Can access everything public, friendly, and protected.**
  - \* **Also has additional data and methods.**
- **A subclass can override a method in the superclass.**
  - \* **Replace inherited method with method specific to subclass.**
- **Static methods can be overridden in a subclass.**

## Single Inheritance

```
public class Account {
 . . .
 // Calculate the interest based on the current interest rate
 public void interest()
 {
 balance = balance + (balance * currentRate); }
 . . .
}

public class CheckingAccount extends Account {
 . . .
 // Override the interest method inherited from Account so
 // interest is only added if balance is over the minimum.
 public void interest()
 { if (balance > minimumBalance)
 { super.interest(); }
 }
 . . . }
```

## Polymorphism

- **Up to now, compiler could bind any call to its exact implementation.**
- **Binding time: The time when the “meaning” of a construct is determined.**
  - \* **Compile time: When the construct is first encountered.**
  - \* **Link time: When results of several compilations are combined.**
  - \* **Execution time: When a program is executed.**
- **Statically typed language: Types are associated with identifiers (variables).**
  - \* **Types matched to identifiers via declarations.**
  - \* **e.g., Java, C++, Eiffel, Object Pascal.**

# Polymorphism

- **Dynamically typed language: Types bound to values not identifiers.**
  - \* “X contains an integer”.
  - \* Cannot assign a variable an illegal value.
  - \* e.g., Smalltalk, Objective C.
- **Most statically typed languages support some dynamic binding for flexibility.**
  - \* The type of a variable can change at run-time.
  - \* Dynamic binding makes it virtually impossible for compiler to bind all calls to implementation at compile time.
- **In statically typed languages, there are rules defining how variables become associated with objects of different types.**

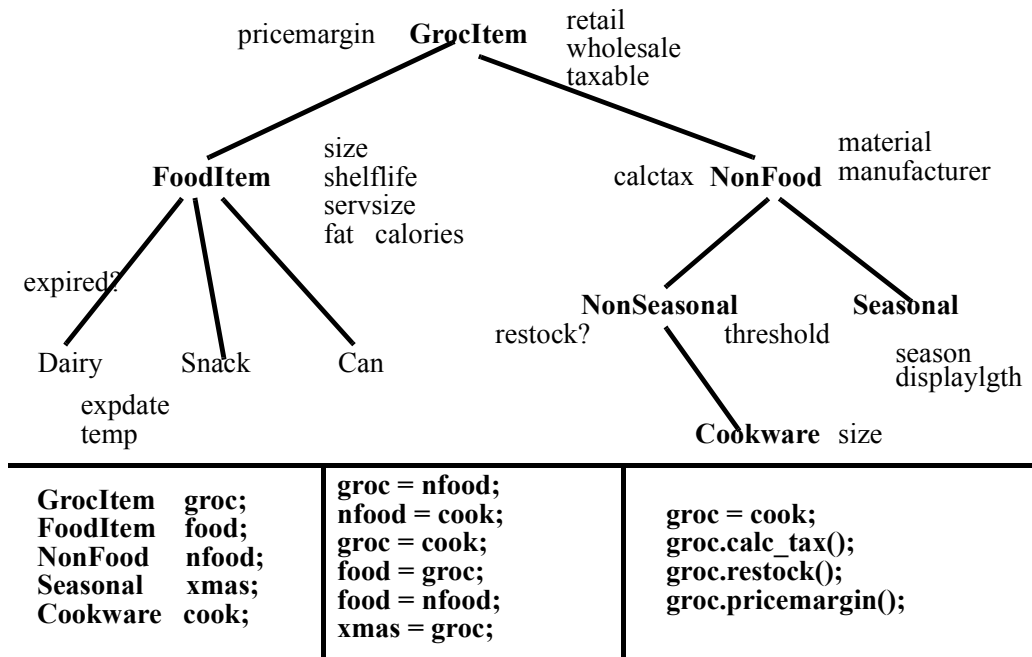
- **In OO languages, these rules usually involve inheritance.**
- **Goals in a statically typed language that supports dynamic binding are:**
  1. **Guarantee at compile time that at least one implementation of each called feature exists.**
  2. **Dynamically bind the call to the most appropriate implementation if there is more than one.**
- **A common way for a variable to change type is by assignment.**
- **Rule:  $x = y$  is legal if and only if  $x$ 's compile-time type is the same as or an ancestor of  $y$ 's compile-time type.**
- **Note that an instance of a child class is a representative of the parent class.**

# Polymorphism

- **Polymorphism:** Ability of a single variable to reference a variety of different objects based on run-time context.
  - \* Instance of class may reference a variable of that type or any of its subtypes.
  - \* Either via assignment or via parameter passing.
- **Polymorphic:** “Many forms.”
- **Most OO programming languages use dynamic binding to determine type of variable within the inheritance hierarchy.**

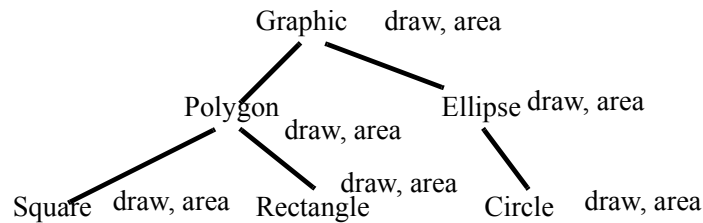
```
Account a = new Account(. . .);
CheckingAccount ca = new CheckingAccount(. . .);
a = ca; // This is legal
a.deposit(200.00); // This is also legal
ca = a; // This is NOT legal.
```

## Quiz: Which are legal??



# Polymorphism

- **If function *f* is expecting a variable of type *A*, it will accept anything that is a kind of *A*.**



- **Suppose we have an array of Graphic objects.**
- **Now lets assume that we fill the array with instances of Square, Rectangle, and Circle.**
- **Each class has its own unique implementation of area method.**

```

Graphic myObjs[] = new Graphic[10];
Square s = new Square(. . .);
Rectangle r = new Rectangle(. . .);
Circle c = new Circle(. . .);
myObjs[0] = s;
myObjs[1] = r;
myObjs[2] = c;
for (i = 0; i < 3; i++)
 { System.out.println("Area: " + myObjs[i].area()); }

```

# Polymorphism

- **The important thing is not that the area of each object is output.**
- **The important thing is that if the first element of the array is a square, then Square's version of area is executed.**
- **Without polymorphism, we are reduced to doing something like this:**

if myobjs[i] is a Square

```
{ System.out.println("Area: " + s.area()); }
```

else if myobjs[i] is a Rectangle

```
{ System.out.println("Area: " + r.area()); }
```

else if myobjs[i] is a Circle

```
{ System.out.println("Area: " + c.area()); }
```

- **Above code replaces ONE line of polymorphic version.**

- **Not only is the code ugly, but it is likely to require frequent maintenance.**
  - \* **What if we want to add a Triangle class?**
  - \* **What is the impact on both versions?**
- **Since static methods are associated with types not objects, a static method is never polymorphic.**

## **Inheritance: Benefits**

### **1. Software Reusability.**

- \* Inherit rather rewrite code.**
- \* Provides greater reliability of code.**
- \* Decreased maintenance costs.**

### **2. Code Sharing.**

- \* Two or more classes inherit from single parent.**
- \* Code used by all subclasses only need be modified in parent class.**

### **3. Consistency of interface.**

- \* All classes that inherit from single parent guaranteed similar behavior.**

## **Inheritance: Benefits**

### **4. Software Components.**

- \* Allows commercial libraries to be constructed and utilized.**

### **5. Rapid prototyping.**

- \* Reusable components reduce time spent on familiar portions of system.**

### **6. Polymorphism.**

- \* Allows high-level components to be generated that can be tailored to fit different applications.**

### **7. Information Hiding.**

- \* Program only needs to know WHAT parent provides, not HOW.**

## **Inheritance: Costs**

### **1. Execution Speed.**

- \* Inherited routines often slower.**
- \* Increase in speed balanced by decrease in development time.**

### **2. Program size.**

- \* Use of any library increases size.**
- \* Less of an issue as memory costs decrease.**
- \* Typically more important to keep down development costs.**

## **Inheritance: Costs**

### **3. Message passing overhead.**

- \* Calling a routine is more costly than simple procedure invocation.**
- \* May be run-time determination.**
- \* Additional cost usually marginal (10%).**
- \* High in dynamically bound languages.**

### **4. Program complexity.**

- \* Overuse of inheritance can increase rather than decrease complexity.**

## Abstract Classes

- **Lets take another look at the Graphic class.**
- **The behavior that is common to all Graphic figures includes:**
  - draw() // Draw the figure on screen
  - area() // Return the area of the figure.
  - rotate() // Rotate the figure
- **None of the above can be implemented in the Graphic class itself.**
  - \* **The implementation is quite different for a Square and a Circle.**
- **Graphic is an example of an abstract class.**
  - \* **Contains important portion of class interface.**

- **Abstract class: A class that has no instances but defines the interface for its descendents.**
  - \* **Defines an abstract concept.**
  - \* **Written with the expectation that its concrete subclasses will add to its structure and behavior.**
  - \* **Used to capture commonalities in the inheritance structure thus making the entire structure easier to understand.**
  - \* **Facilitates reuse.**
- **Can't instantiate an abstract class.**
- **Allows objects of the various child types to be treated identically by calling the same method on the objects.**
- **Abstract classes have some methods implemented and some not.**

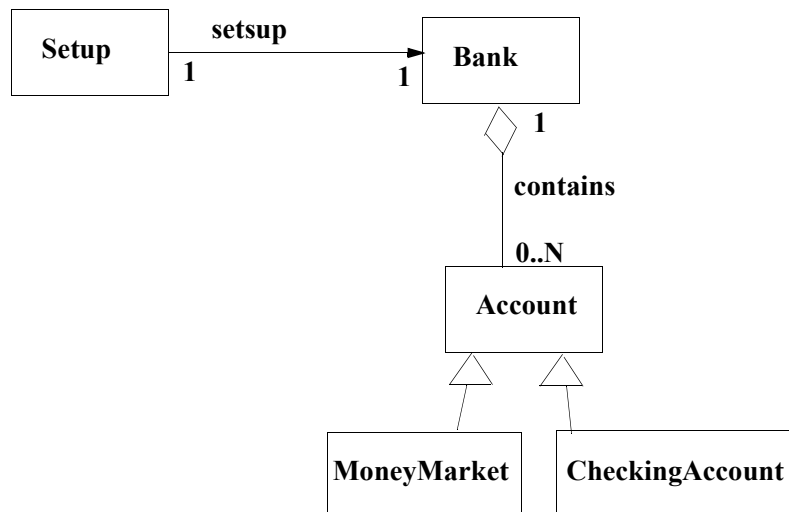
## Abstract Classes

- Those that are implemented do not need to be reimplemented in subclasses.
- The higher up an implementation is placed, the greater its potential for reuse.
- Abstract classes play a role in polymorphism.
- In Java, abstract classes declared using keyword `abstract`:

```
abstract public class Account {
 . . . }
```

- Abstract classes must contain at least one abstract method.
- Concrete class that inherits from an abstract class must override every abstract method of superclass.

## Account Example



```

// This is an abstract class. All subtypes of this class
// must implement the abstract methods of "interest".
public abstract class Account {

 // Attributes for account number, current balance and current interest rate.
 protected int acctNumber;
 protected double balance;
 protected static double currentRate = 0.02;

 // Constructor that sets the account number.
 public Account(int ano)
 { acctNumber = ano; }
 // Overloaded constructor that takes both an account
 // number and an initial balance for the account.
 public Account(int ano, double bal)
 {
 acctNumber = ano;
 balance = bal;
 }
}

```

```

// Method to allow a positive amount to be deposited into the account.
public void deposit(double amt)
{
 if (amt > 0.0)
 { balance += amt; }
}
// Method to allow a positive amount to be withdrawn from the account.
public void withdraw(double amt)
{ if (amt >= 0.0)
 { balance -= amt; }
}
// Method to retrieve the current account balance.
public double getBalance()
{ return balance; }

// Static method to set the rate.
public static void setRate(double newrate)
{ currentRate = newrate; }
public static double getRate()
{ return currentRate; }
}

```

```

public void displayAccount()
{
 System.out.println("Account number: " + acctNumber);
 System.out.println("Account balance: " + balance);
 System.out.println();
}
// ***** ABSTRACT METHODS *****
// Calculate the interest
public abstract void interest();

// Calculate any bank charges
public abstract void charges();

}

```

```

// File CheckingAccount.java
// Author: Heidi Ellis
// Creation date: 7/6/99
// Supertype: Account
// Purpose: Represents a checking account that is a specialized type of
// Account. Checking accounts have a minimum balance of $500.00

public class CheckingAccount extends Account {
 // Attributes to hold the minimum balance.
 protected static double minimumBalance = 500.00;
 protected static double charge = 10.00;
 protected static currentRate = 0.02;

 // Constructor that takes an account number and initial
 // balance, calls superclass constructor and checks initial balance.
 public CheckingAccount(int actno, double bal)
 {
 super(actno, bal);
 if (bal < minimumBalance)
 { System.out.println("Error: Insufficient initial balance."); }
 }
}

```

```

// Implement the abstract "interest" method defined in Account so that
// interest is only added if the balance is over the minimum.
public void interest()
{
 if (balance > minimumBalance)
 { balance = balance + (balance * currentRate); }
}

// Implement the abstract "charges" account defined in Account so
// so that a charge is levied if the balance is below the minimum.
public void charges()
{
 if (balance < minimumBalance)
 { balance = balance - charge; }
}
}

```

```

// File MoneyMarket.java
// Author: Heidi Ellis
// Creation date: 7/7/99
// Supertype: Account
// Purpose: Represents a money market account that is a specialized type of
// Account. Money markets have a minimum balance of $10000.00

public class MoneyMarket extends Account {

 // Attributes to hold the minimum balance.
 protected static double minimumBalance = 10000.00;
 protected static double bonusBalance = 500000.00;
 protected static double charge = 100.00;
 // Constructor that takes an account number and initial
 // balance, calls superclass constructor and checks initial balance.
 public MoneyMarket(int actno, double bal)
 { super(actno, bal);
 setRate(10.0);
 if (bal < minimumBalance)
 { System.out.println("Error: Insufficient initial balance."); }
 }
}

```

```

// Implement the abstract "interest" method defined in Account so that
// interest is only added if the balance is over the minimum.
public void interest()
{ if (balance > minimumBalance)
 { if (balance > bonusBalance)
 { balance = balance + (balance * 1.5 * currentRate); }
 else balance = balance + (balance * currentRate);
 }
}

// Implement the abstract "charges" account defined in Account so
// so that a charge is levied if the balance is below the minimum.
public void charges()
{ if (balance < minimumBalance)
 { balance = balance - charge; }
}
}

```

```

// File: Bank.java
// Author: Heidi Ellis
// Creation date: 6/24/99
// Purpose: This class represents a general bank. It contains a
// maximum interest rate to be applied to accounts and
// a series of accounts.

public class Bank
{
 // The accounts are held in an array.
 protected Account theAccounts[];
 // Keep track of the current number of existing accounts
 protected int noAccounts;

 // This constructor sets up the array to hold the accounts.
 // For simplicity, we'll assume a maximum of 20 accounts.
 public Bank()
 { theAccounts = new Account[20];
 noAccounts = 0; }
}

```

```

public void newCheckingAccount(double amt)
{ // Create a new checking account.
 CheckingAccount myChecking = new CheckingAccount(noAccounts, amt);
 // Add the account to the Bank's collection
 theAccounts[noAccounts] = myChecking;
 noAccounts++;
}

public void newMoneyMarketAccount(double amt)
{ // Create a new money market account.
 MoneyMarket myMoney = new MoneyMarket(noAccounts, amt);
 // Add the account to the Bank's collection
 theAccounts[noAccounts] = myMoney;
 noAccounts++;
}
// This function prints all Accounts
public void displayBankAccounts()
{ System.out.println();
 for (int i = 0; i < noAccounts; i++)
 { theAccounts[i].displayAccount(); }
}

```

```

public void calculateInterest()
{ for (int i = 0; i < noAccounts; i++)
 { theAccounts[i].interest(); }
}
}

```

```

// File: Setup.java
// Author: Heidi Ellis
// Creation date: 6/22/99
// Purpose: This is class sets up an account and makes some
// method calls on the Account object.
public class Setup
{
 public static void main(String[] arg)
 {
 Bank theBank;
 theBank = new Bank();
 // Tell the bank to create several accounts with various initial balances.
 theBank.newCheckingAccount(2000);
 theBank.newMoneyMarketAccount(20000);
 theBank.newMoneyMarketAccount(600000);
 theBank.newCheckingAccount(4000);
 theBank.displayBankAccounts();
 theBank.calculateInterest();
 theBank.displayBankAccounts();
 } // End the main function.
}

```

```

Account number: 0
Account balance: 2000.0

Account number: 1
Account balance: 20000.0

Account number: 2
Account balance: 600000.0

Account number: 3
Account balance: 4000.0

Account number: 0
Account balance: 2040.0

Account number: 1
Account balance: 220000.0

Account number: 2
Account balance: 9600000.0

Account number: 3
Account balance: 4080.0

```

## Final Classes

- **Occasionally you want to prevent someone from deriving a class from an existing class**
- **Java uses the final modifier to indicate that subtyping off of a class is not allowed.**

```
final class MoneyMarket { . . . }
```

- **You can also make a specific method in a class final.**

```
final void calcMyInterest() { . . . }
```

- \* **No subclass can override the calcMyInterest method.**
- \* **All methods in a final class are automatically final.**

- **A class or method is made final because:**

1. **Efficiency: Final methods use static binding.**
2. **Safety: Ensures that the method you defined is called.**
3. **Complexity: Limits size of inheritance hierarchy.**

## Determining Type

- **Just as variables of primitive types can be cast, variables that reference objects can also be cast.**

**\* Done when you need to access method defined on subtype and not on supertype.**

```
CheckingAccount ca = (CheckingAccount)accounts[0];
```

- **But what if you don't know the exact type of accounts[0]?**
- **It is good programming practice to discover the type of a variable before casting.**
- **Java provides the instanceof operator.**

**\* Returns true if object to left of operator is same type as object to right.**

```
if (accounts[0] instanceof CheckingAccount)
 { ca = (CheckingAccount)accounts[0]; }
```

**\* Because inheritance means “is-a”, instanceof will also return true if object to the left is a subtype of the object on the right.**

- **The instanceof operator is used frequently to ensure proper casting.**

## class Object

- **All Java classes implicitly inherit from java.lang.Object**
  - \* extends keyword not required
  - \* provides methods which all objects have
- **public boolean equals(Object obj) -  $x == y$  means identity; returns true if x and y are same object. x.equals(y) means equivalence; returns true if x and y have same value**
  - \* May be implemented by subclass
  - \* Default implementation uses identity as criteria
- **public int hashCode() - returns a unique value to allow objects to be stored efficiently in a Hashtable collection.**
  - \* A subclass which overrides equals should also override hashCode so that equal objects return the same hash code.

- **protected void finalize() - called by garbage collector before returning object to heap.**
- **protected Object clone() - creates a new object which is a copy of the existing one. Must be implemented by the subclass or an exception is thrown**
- **public final Class getClass() - returns a Class object which contains information about the type of the object**
- **public String toString() - returns a String which describes the object**
  - \* Default implementation is somewhat ugly
  - \* Recommended that each subclass should override
  - \* Note: following are equivalent

```
System.out.println(x);
System.out.println(x.toString());
```

## class Object

- wait, notify, and notifyAll **support multi-threaded programming.**

## Substitutability and Inheritance

- **Any object of a derived class must be able to be substituted in place of a parent class.**
- **Why?**
- **“Liskov substitution principle” - Barbara Liskov**

# Exceptions and Inheritance

- **In order to ensure substitutability and common interface, exceptions must be inherited.**
  - \* **Guarantees behavior of a class is compatible with ancestors.**
- **In Java, subclasses cannot add new exceptions to method interface when overriding**
  - \* **May add subclasses of existing exceptions.**
- **Redefinition of a method:**
  1. **Postcondition can only be made “stronger”.**
    - \* **More difficult to satisfy.**
    - \* **Subclasses guaranteed to perform at least all functions of superclasses.**

2. **Precondition can only be made “weaker”.**
  - \* **Easier to satisfy.**
  - \* **Subclasses guaranteed to perform in at least all cases their superclasses would accept.**