CSC 102 Lecture Notes Week 4
Converting between Java Types
Inheritance

I. **Relevant reading.**

A. Horstmann Chapter 10, Sections 10.1 - 10.7

B. Horstmann Chapter 9, Sections 9.1 - 9.3 (again)

II. **Converting between class and interface types.**

A. Classes and interfaces define *types*.

B. An object of one type can be converted to another with a *cast*.

1. The syntax of a cast in Java is the same as it is in C:
   a. The name of a type is enclosed in parentheses and put in front of a variable or other value to be cast.
   b. For example:

```
Shape s = new Rectangle;       // Put a rectangle into a general Shape variable
Rectangle r = (Rectangle) s;   // Put that object into a Rectangle variable
```

2. Casts are not always legal, since not all types can be meaningfully converted to each other.
   a. It's good practice to check that what you're trying to cast is legal.
   b. For example, the following is not legal

```
Shape s = new Rectangle;
Circle c = (Circle) s;
```

   c. Understanding what's legal in the way of casting is the subject of Lab 7.

C. You can determine the type of an object using the **`instanceof`** operator.

1. It produces a boolean value.

2. E.g., given the preceding variable declarations, here are a couple examples of using `instanceof`:

```
r instanceof Shape      // true
r instanceof Circle     // false
```

3. As noted above, it's good practice to check before casting to avoid errors; here's an example of doing this using `instanceof`

```
Shape s = getSomeShape();   // Call a method that returns any shape,
                            // e.g., from a user's input

// Check what type of shape it is, before doing a cast
if (s instanceof Rectangle) {
    System.out.println("width = " + ((Rectangle) s).getWidth());
}
else if (s instanceof Circle) {
    System.out.println("radius = " + ((Circle) s).getRadius());
}
```

D. A method related to instance of is `Object.getClass`

1. It returns the class value of an object, as in

Now on to the new subject of inheritance in Java,
as covered in Horstmann Chapter 10

III. **Introduction to Java inheritance.**

A. In Java, a class can `extend` another class.

B. Such extension defines an *inheritance* relationship.

1. The class being extended is called the *parent* class, or the *superclass* of the inheritance relationship.
2. The class doing the extending is called the *child* class, or the *subclass* of the inheritance relationship.

C. In a UML diagram, inheritance is shown with a hollow triangular arrow, with the arrowhead pointing at the parent class, and solid lines between the child and parent classes.

D. We'll see some examples coming right up.

IV. **An initial example, from book Chapter 10 (Section 10.1).**

A. The book has a good illustration of basic inheritance ideas in its `BankAccount` example

```
import java.text.DecimalFormat;

/****
 *
 * Class BankAccount is a simple banking example based on the example of the
 * same name from Horstmann Chapter 10.  It has a balance that can be changed
 * by deposit and withdrawal methods.  It also provides a method to get the
 * current balance.
 *
 */
public class BankAccount {

    /** The current balance of this bank account. */
    private double balance;

    /**
     *  Construct a bank account with a zero balance.
     */
    public BankAccount() {
        balance = 0;
    }

    /**
     * Construct a bank account with the given initial balance.
     */
    public BankAccount(double balance) {
        this.balance = balance;
    }

    /**
     * Deposit the given amount of money into this bank account.
     */
    public void deposit(double amount) {
        if (amount < 0) {
            throw new IllegalArgumentException();
        }
        balance = balance + amount;
    }
```

```
    /**
     * Withdraw the given amount of money from this bank account.
     */
    public void withdraw(double amount) {
        if ((balance - amount < 0) || (amount < 0)) {
            throw new IllegalArgumentException();
        }
        balance = balance - amount;
    }

    /**
     * Get the current balance of this bank account.
     */
    public double getBalance() {
        return balance;
    }

    /**
     * Transfer the given amount from this account to the given other account.
     */
    public void transfer(double amount, BankAccount other) {
        withdraw(amount);
        other.deposit(amount);
    }

}
```

B. Suppose that we want to define a couple specialized forms of bank account:
   1. A savings account, which earns interest.
   2. A checking account, which charges a transaction fee after a certain number of transactions have occurred in a given period of time.

C. We'll use Java's inheritance to define these two forms of account.

V. **Implementing subclasses (Sections 10.2 - 10.4).**

A. Using inheritance, the preceding forms of account can be implemented as follows:

```
/****
 *
 * A SavingsAccount extends a BankAccount by adding functionality for an
 * account to earn interest.  A SavingsAccount has an interest rate, with a
 * method to compute the interest and add it to the account balance.
 *
 */
public class SavingsAccount extends BankAccount {

    private double interestRate;

    /**
     * Construct a savings account with the given interest rate.
     */
    public SavingsAccount(double rate) {
        interestRate = rate;
    }

    /**
     * Add the earned interest to this account.
     */
    public void addInterest() {
```

```
            double interest = getBalance() * interestRate / 100;
            deposit(interest);
        }

    }

    /****
     *
     * A CheckingAccount extends a BankAccount by adding functionality for
     * transaction fee charges.  CheckingAccount specializes the BankAccount
     * deposit and withdraw methods to perform transaction counting.
     * CheckingAccount also provides , and a method to deduct the chareges from the
     * account.
     */
    public class CheckingAccount extends BankAccount {

        /** Number of transactions before transaction fees start */
        private static final int FREE_TRANSACTIONS = 3;

        /** Dollar amount of transaction fee (what a rip off) */
        private static final double TRANSACTION_FEE = 2.0;

        /** Count of transactions since last fee deduction */
        private int transactionCount;

        /**
         * Construct a checking account with the given balance.
         * @param initialBalance the initial balance
         */
        public CheckingAccount(double initialBalance) {

            // Construct the superclass.
            super(initialBalance);

            // Initialize transaction count in this subclass.
            transactionCount = 0;
        }

        /**
         * Deposit the given amount of money into this checking account and
         * increment the transaction count by 1.
         */
        public void deposit(double amount) {

            // Call the parent class deposit method to update the balance.
            super.deposit(amount);

            // Increment this transaction count
            transactionCount++;
        }

        /**
         * Withdraw the given amount of money from this checking account and
         * increment the transaction count by 1.
         */
        public void withdraw(double amount) {

            // Call the parent class withdraw method to update the balance.
            super.withdraw(amount);
```

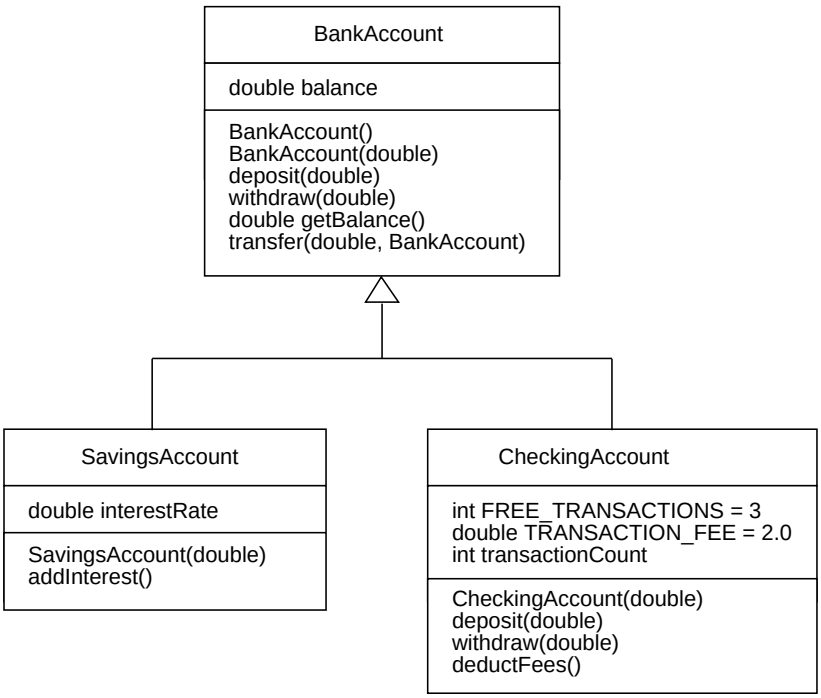
```
            // Increment this transaction count
            transactionCount++;
        }

       /**
        * Deduct the accumulated fees and reset the transaction count.
        */
        public void deductFees() {
            if (transactionCount > FREE_TRANSACTIONS) {
             double fees = TRANSACTION_FEE *
                   (transactionCount - FREE_TRANSACTIONS);
             super.withdraw(fees);
          }
          transactionCount = 0;
       }

        //
        // Note that getBalance is not specialized, since it works the same for a
        // checking account as for a regular bank account.
        //

   }
```

B. Here is a UML diagram of the three classes:



C. Some important observations about this inheritance hierarchy:

1. Both `SavingsAccount` and `CheckingAccount` inherit `BankAccount.getBalance` and `BankAccount.transfer` without overriding them; this means that `getBalance` and `transfer` are automatically defined for both subclasses.
2. `SavingsAccount` does not override `deposit` or `withdraw`, which means these two methods are also automatically defined for `SavingsAccount`.
3. `CheckingAccount` does override `deposit` and `withdraw`, which means these two methods are

specialized for `CheckingAccount`; but note that the specialized implementations call `super`, which uses `BankAccount`'s implementation, plus adds some additional behavior.

4. Both subclasses inherit the `BankAccount.balance` data field; however, since that field is `private` in `BankAccount`, the subclasses cannot access it directly, but can access it indirectly by calling the parent class methods.
5. Both subclasses add one or more private data fields of their own, which exist in the subclasses, but not in the parent class.
6. Both subclasses define an initializing constructor, which they implement in different ways.

VI. **A careful look at the subclass constructors.**

```
public CheckingAccount(double initialBalance) {

    // Construct the superclass.
    //
    // NOTE: We explicitly call the parent constructor via super.  This means
    //       the initial balance is set.
    //
    super(initialBalance);


    // Initialize transaction count in this subclass.
    //
    // NOTE: Since transactionCount is a non-inherited data field, we need to
    //       do its initialization explicitly here (there's no way that the
    //       parent class can initialize it).
    //
    transactionCount = 0;

}

public SavingsAccount(double rate) {

    //
    // Initialize the non-inherited data field.
    //
    // NOTE: In contrast to the CheckingAccount constructor, this constructor
    //       does not explicitly call super.  This means that the default
    //       parameterless version of the parent constructor is automatically
    //       called.  This is a general rule of Java.  I.e., when a child
    //       constructor does not explicitly call its parent constructor via
    //       super, Java calls the default version of the parent constructor on
    //       the child's behalf.  In such cases, the parent class MUST define a
    //       default constructor, otherwise the compiler gives an error.
    //
    interestRate = rate;

}
```

VII. **A careful look at `CheckingAccount.deposit`.**

```
public void deposit(double amount) {

    //
    // Call the parent class deposit method to update the balance.
    //
    // NOTE: Calling this.deposit here is a big mistake (see Page 428); we need
    //       to use super.deposit to access the parent version of the deposit
```

```
        //       method.
        //
        // NOTE ALSO: We cannot use the following in this version of deposit
        //            because balance is a private data field of BankAccount:
        //
        //                balance = balance + amount;
        //
        //            The balance field is inherited by CheckingAccount, but its
        //            name is not visible due to its private protection.  Hence,
        //            we need to call the public method BankAccount.deposit to
        //            effect the change to balance.
        //
        super.deposit(amount);

        //
        // Increment this transaction count.
        //
        // NOTE: Here we are referring to this.transactionCount, since there is no
        //       such data field in the parent class.
        //
        transactionCount++;
    }
```

VIII. **Here's a simple `BankAccount` tester class, a la Section 10.6 of the book.**

```
/****
 *
 * This is a simple testing program for the chapter 10 examples of BankAccount
 * and its subclasses.  This program includes code from the book's
 * AccountTester class, plus some additional code to discuss during 102
 * lecture.
 *
 */
public class BankAccountChapter10Tester {

    public static void main(String[] args) {

        /*
         * Create a savings and checking account.
         */
        SavingsAccount momsSavings = new SavingsAccount(0.5);
        CheckingAccount harrysChecking = new CheckingAccount(100);

        /*
         * Make a savings deposit.
         */
        momsSavings.deposit(10000);

        /*
         * Transfer some funds, then withdraw.
         */
        momsSavings.transfer(2000, harrysChecking);
        harrysChecking.withdraw(1500);
        harrysChecking.withdraw(80);

        /*
         * Transfer and withdraw some more.
         */
        momsSavings.transfer(1000, harrysChecking);
        harrysChecking.withdraw(400);
```

```
/*
 * Compute interest for the savings account, deduct fees for the
 * checking.
 */
momsSavings.addInterest();
harrysChecking.deductFees();

/*
 * Print some results.
 */
System.out.println();
System.out.println("Mom's savings balance: "
      + momsSavings.getBalance());
System.out.println("Expected:              7035.0");
System.out.println();

System.out.println("Harry's checking balance: "
      + harrysChecking.getBalance());
System.out.println("Expected:                 1116.0");

/*
 * Here are some illustrations of polymorphism.
 */
BankAccount momsSavingsB = momsSavings;
Object momsSavingsO = momsSavings;
System.out.println();
System.out.print("Savings balances: " +
    ((SavingsAccount)momsSavingsO).getBalance() + " ");
System.out.println(((SavingsAccount)momsSavingsO).getBalance());
System.out.println("Expected:         7035.0 7035.0");


/*
 * Here are some calls that generate errors.  Convince yourself that
 * you understand why.  (They're commented out so the program will
 * compile.)
 */
// SavingsAccount emptySavings = new SavingsAccount();
// CheckingAccount emptyChecking = new CheckingAccount();
// momsSavings = (SavingsAccount) harrysChecking;
// CheckingAccount momsChecking = (CheckingAccount) momsSavings;
BankAccount mom2 = momsSavings;
// CheckingAccount momsChecking = (CheckingAccount) mom2;


/*
 * An important point of method overriding is that the appropriate
 * version of an overridden method be called.  For example, when we
 * call deposit on a CheckingAccount object, we want to be sure to get
 * the version of deposit declared in CheckingAccount, not the more
 * general version in BankAccount.  And we want this even if we are
 * referring to a CheckingAccount from a BankAccount variable.
 *
 * HOWEVER, suppose we'd like to call the general version of a method
 * explicitly?  How could we do this?
 */
BankAccount checking = (BankAccount) new CheckingAccount(100);
checking.deposit(100);  // Which deposit are we calling here?

((BankAccount) checking).deposit(200);  // What about here?
```

```
    }

}
```

IX. Updating BankAccount to Implement the Comparable Interface

A. The `Comparable` interface is used in Java when a objects need to be comopared in meaningful ways.

B. For example in Program 3, you are asked to have the absract `Shape` class implement `Comparable` so the two shapes can be compared.

C. There are number of Java libray classes that rely on the fact that a class implements Comparable, in particular class that provide `sorting` methods.

D. Here's an example of how the `BankAccount` class can implement `Comparable`:

```
/****
 *
 * This is a version of the BankAccount class that implements the Comparable
 * interface.  It has exactly two differences with the original definition of
 * BankAccount:
 *
 *   (1) implementation of Comparable<BankAccount>
 *   (2) addition of the compareTo method
 *
 * See lecture notes Week 4 for some discussion.
 *
 */
public class BankAccount implements Comparable<BankAccount> {

    . . .

    /**
     * Compare this account with the given other account, using balance as the
     * basis of comparison.
     */
    public int compareTo(BankAccount other) {
        if (balance < other.balance) return -1;
        if (balance == other.balance) return 0;
        return 1;
    }

}
```

1. By implementing `compareTo`, a list of bank accounts can be sorted using the Java library method `Collections.sort`.
2. An example of this is illustrated in the enhanced bank account examples discussed next.

X. Enhnaced bank account examples.

A. The `102/examples` web page has some further enhancments to the `BankAccount` class, along with some additional classes that illustrate the uses of inheritance and interfaces.

B. We'll discuss these examples in class, and you can view the source code for details.

XI. Polymorphsim revisited (Sections 9.3 and 10.6).

A. In last week's notes, we defined a polymorphic method as one that takes parameters of an interface type.

B. Now we can update the definition to say that a polymorphic method is any method that has one or more of the following properties:

1. it takes one or parameters of an interface type
2. it takes one or parameters of an extended class type, i.e., a class that has been extended at least once
3. it is defined in an implemented interface or an extended class

C. For example, `BankAccount.transfer(double, BankAccount)` is polymorphic in its second parameter, which may be sent objects of type `BankAccount`, or its two subtypes.

D. `BankAccount.getBalance()` is polymorphic because it is defined in an extended class, and can be invoked from objects of type `BankAccount`, `SavingsAccount`, or `CheckingAccount`

XII. **Access Control (Section 10.7).**

A. For now in 102, the following rules remain in force:
1. Methods that are to invoked outside the class are `public`.
2. All data fields are `private`.
3. All methods that are not invoked from outside of the class are `private`.

B. We will cover this subject further as the quarter progresses.

XIII. **`Object`: The book's treatment of the Cosmic Superclass (Section 10.8).**

A. `java.lang.Object` is the class from which all other Java classes inherit.

B. Whenever you declare a class, it automatically extends `Object`.

C. As we've discussed in past lectures, `Ojbect` has a number of methods that are considered universally applicable to all objects, including,
1. `String toString()` -- *Returns a string representation of the object*; remember that `toString` does not print anything itself, it just returns a string that can be printed
2. `boolean equals(Object otherObject)` -- *Tests whether the object equals another object*
3. `Object clone()` -- *Makes a full copy of an object*

D. The book suggests that it's a good idea to override these methods in all of your classes, which it often is.
1. ***HOWEVER,*** you do not need to define these methods for the classes you write in CSC 102, unless a writeup explicitly instructs you to do so.
2. Sections 10.8.1 through 10.8.3 of the book describe in detail how the overrides are implemented, and discusses related issues; it's worth a read.
3. We'll return to this subject matter in upcoming lectures.

XIV. **Some Special Topics from the book, which you should read.**

A. ***Abstract classes (Special topic 10.1)***

B. ***Final methods and classes (Special topic 10.2)***

C. ***Protected access (Special topic 10.3)***; but remember that for now we'll ***not use*** protected access in 102 labs or programs.

XV. **Here's a summary of the differences between interfaces and abstract classes.**

	Interface	**Abstract Class**
keyword	`interface`	`abstract class`

inherits-from keyword	`implements`	`extends`
data fields	none	allowed
methods	all abstract	can be declared `abstract`
method bodies	none	allowed
multiple inheritance	yes	no

XVI. **Some design alternatives for bank accounts.**

A. Here are some questions regarding the way `BankAccount` and its subclasses have been designed so far, and how that design could be altered or carried further:

1. When the `BankAccount` class implements the `Valuable` interface, does it mean that its subclasses inherited the implementation?
2. Could `BankAccount` have been defined as an interface instead of a class? If so, what how would the extending classes have to be changed into implementing classes?
3. Could `BankAccount` have been defined as a (partially) abstract class? If so, what methods would it make sense to define as abstract?

B. You should think about the answers to these questions.