Inheritance

- All classes inherit from the `Object` class in Java.
- All classes are arranged in a hierarchy (a tree, more about this later in the semester) with `Object` at the top of the hierarchy.
- A class that inherits from another is called a subclass.
- A class that provides attributes and methods for inheritance by subclasses is called a superclass.
Inheritance keywords

- A class that **extends** another class is a subclass that inherits all fields and methods of the superclass.
- The subclass has direct access to all fields that are **public** and **protected**.
- The subclass can override the definitions of inherited methods with new implementations (using the same signature) and can access overridden methods using the **super** keyword.
Example: A superclass

```java
public class BankAccount {
    private double balance;

    public BankAccount() {...}
    public BankAccount(double initBalance) {...}
    public void deposit(double amount) {...}
    public void withdraw(double amount) {...}
    public double getBalance() {...}
}
```

Example: A subclass

```java
public class SavingsAccount extends BankAccount {
    private double interestRate;
```
Constructors

```java
public SavingsAccount(double initRate, double initBalance) {
    super(initBalance);
    interestRate = initRate;
}

public SavingsAccount(double initRate) {
    super();
    interestRate = initRate;
}
```

New method

```java
public void addInterest() {
    deposit(getBalance() * interestRate);
}
```
Another way: protected

In BankAccount:
\[
\text{protected double balance;}
\]

In SavingsAccount:
\[
\text{public void addInterest() \{
    deposit(balance*interestRate);
\}}
\]

A field (or method) with protected visibility can be accessed in either the class that defines it, its subclasses or any class in the same package.

Overriding

- A savings account subtracts a fee of $10 for withdrawals over $1000.
- We need to override the withdraw method and provide a new implementation that is appropriate.

\[
\text{public void withdraw(double amount) \{
    if (amount > 1000.0)
        super.withdraw(amount + 10.0);
    else
        super.withdraw(amount);
\}}
\]
Overriding Object methods

Classes inherit methods from the `Object` class. These methods typically do not work properly for our specific subclasses so we must override them.

```java
public boolean equals(Object obj) {
    SavingsAccount other = (SavingsAccount)obj;
    return
        this.interestRate == other.interestRate
        && this.balance == other.balance;
}
```

(assuming `balance` is protected in the `BankAccount` class)

**Methods you should override:**

- `public boolean equals(Object obj)`
  - Compares this object with the specified object by comparing the references only.
- `public String toString()`
  - Returns the class name + "@" + the hexadecimal representation of the object's hashcode.
- `public int hashCode()`
  - Calculates the hashcode of this object based on its reference only.
Polymorphism

BankAccount acct;
acct = new BankAccount(15111.0);
acct.withdraw(2000.0);
System.out.println(acct.getBalance());
acct = new SavingsAccount(15111.0);
acct.withdraw(2000.0);
System.out.println(acct.getBalance());

The same statement calls two different methods.

The Stack<\text{E}> class

- The \texttt{java.util} package includes a \texttt{Stack<\text{E}>} class.
- \texttt{Stack<\text{E}>} includes these \texttt{public} methods:
  - \texttt{boolean empty()}
  - \texttt{E peek()}
  - \texttt{E pop()}
  - \texttt{E push(E item)}
  - \texttt{int search(Object obj)}
- \texttt{Stack<\text{E}>} extends \texttt{Vector<\text{E}>}.
  - Why is this very poor design decision?
Interfaces

- A Java interface (not a GUI) is a means for defining specifications for behaviors that are common across classes that are not directly related by inheritance.

```java
public interface Comparable<T> {
    int compareTo(T obj);
}
```

An interface cannot be instantiated directly:

```java
Comparable<String> s = new Comparable<String>();
```

Using Interfaces

Any class that provides the behavior(s) specified in an interface must implement that interface and provide implementations for the methods specified.

```java
public class BankAccount implements Comparable<BankAccount> {
    ...
    public int compareTo(BankAccount other) {
        ... // provide implementation
    }
}
```
The **Queue<E>** interface

- The `java.util` package includes a `Queue<E>` interface.
- `Queue<E>` includes specifications for these methods:
  - `boolean offer(E item)`
  - `E element()` *
  - `E peek()` **
  - `E remove()` *
  - `E poll()` **

  * Throws an exception if queue is empty.
  ** Returns null if the queue is empty.

Using **Queue<E>**

- `LinkedList<E>` implements `Queue<E>`.
- Example: A queue of customers

```java
Queue<Customer> customerQ =
    new LinkedList<Customer>();
Customer c = new Customer("Andrew");
customerQ.offer(c); // OK
customerQ.add(c);   // OK
customerQ.addFirst(c); // BAD (Why?)
```
Abstract Classes

- An abstract class can have abstract methods like interfaces and also fields.
- Abstract classes can also have concrete (implemented) methods.
- An abstract class cannot be instantiated directly.
- Subclasses of an abstract class must provide implementations of the abstract methods specified in the abstract class definition.
- Abstract classes can have constructors that initialize fields defined in the abstract class.

Abstract Class Example

```java
public abstract class Vehicle {
    private int speed;
    private String manufacturer;
    ...
    public int getSpeed { return speed; }
    public String getManufacturer {
        return manufacturer;
    }
    public abstract double computeTax();
    ...
}
```
Abstract Class Example (cont'd)

```java
public class Truck extends Vehicle {
    private int numWheels;
    ...
    public double computeTax() {
        return 10.0 * numWheels;
    }
    ...
}
```

Correct Usage

Interfaces:

```java
String s = new String("Pittsburgh");
Comparable<String> s = new String("Erie");
```

Abstract classes:

```java
Truck myTruck = new Truck("Good Humor");
Vehicle myRide = new Truck("Ryder");
```
Comparison

<table>
<thead>
<tr>
<th></th>
<th>Actual Class</th>
<th>Abstract Class</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instances can be created</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Can define fields and methods</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Can define constants</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Number of these a class can extend</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td>Number of these a class can implement</td>
<td>0</td>
<td>0</td>
<td>any number</td>
</tr>
<tr>
<td>Can extend another class</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Can declare abstract methods</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Can declare variables of this type</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Casting

- In the `equals` method, we used the following casting:
  
  ```java
  SavingsAccount other = (SavingsAccount) obj;
  ```

- What if `obj` isn't a `SavingsAccount`?
  - A `ClassCastException` is generated during runtime.

- We can use the `instanceof` operator to check the object's actual type during runtime.
instanceof example

```java
public boolean equals(Object obj) {
    if (obj instanceof SavingsAccount) {
        SavingsAccount other = (SavingsAccount)obj;
        return
            this.interestRate == other.interestRate
            && this.balance == other.balance;
    }
    return false;
}
```

Copying data

```java
public class Player {
    String name;
    int number;
    CalendarDate birthday;
}

Person p1 = new Person("Roethlisberger", 7,
        new CalendarDate(3, 2, 1982);
```
Copying data

Person p2 = p1;

Cloning: Shallow Copy

In the Person class:

```java
public Object clone() {
    Object newObj = new Person(name, number, birthday);
    return newObj;
}
```
Cloning: Shallow Copy

Person p2 = (Person)p1.clone();

```
name  | "Roethlisberger"
number | 7
birthday

month | 3
day   | 2
year  | 1982
```

Cloning: Deep Copy

In the Person class:

```java
public Object clone() {
    try {
        Person newPerson = (Person)super.clone();
        newPerson.birthday =
            (CalendarDate)birthday.clone();
        return newPerson;
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}
```

Object has a clone method that performs the shallow copy.

CalendarDate must also have a clone method.
Cloning: Deep Copy

In the CalendarDate class:
public Object clone() {
    try {
        CalendarDate newDate =
            (CalendarDate)super.clone();
        return newDate;
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}

Cloning: Deep Copy

The method Object.clone will generate a CloneNotSupportedException if it is called in a class that does not implement the Cloneable interface. Therefore:
public class Person implements Cloneable {
    ...
}
public class CalendarDate implements Cloneable {
    ...
}
Cloning: Deep Copy

Person p2 = (Person)p1.clone();

Strings are immutable

Cloning: Deep Copy