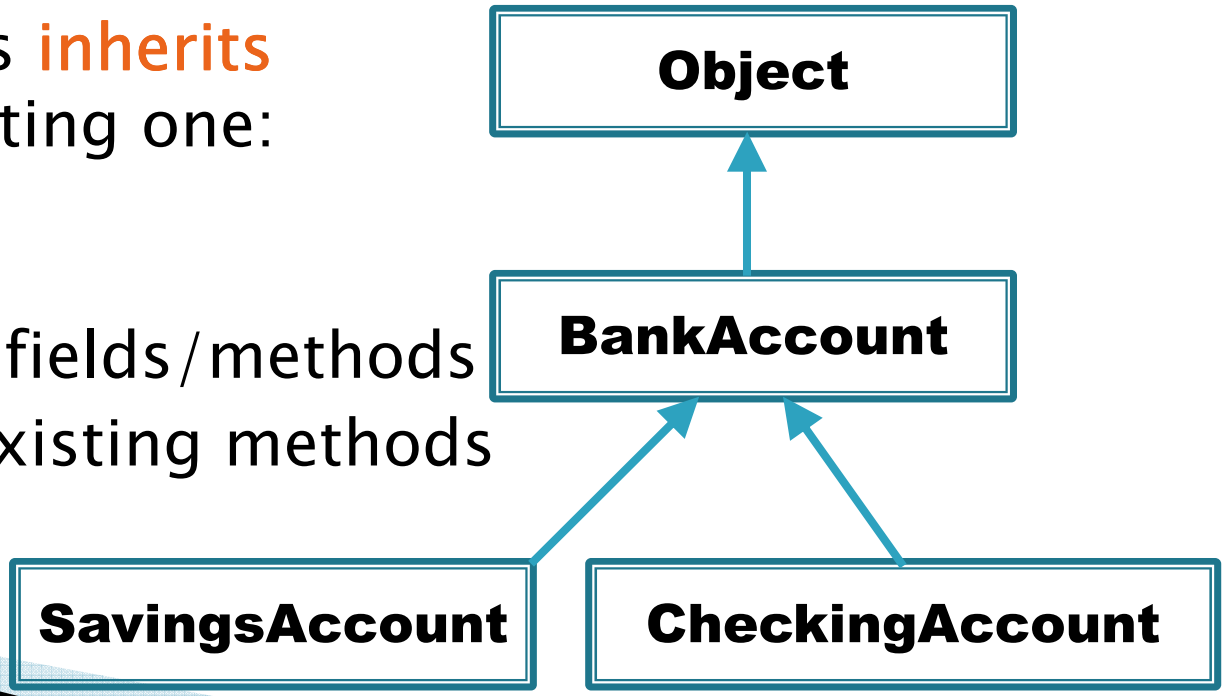
Inheritance – what is it?

- Sometimes a new class is a special case of the concept represented by another
  - A SavingsAccount *is-a* BankAccount
  - An Employee *is-a* Person
- Can extend existing class, changing just what we need
- The new class inherits from the existing one:
  - all methods
  - all fields
- Can add new fields/methods
- Or override existing methods



```
public class BankAccount {
    private double balance;

    public BankAccount() {
        this(0.00);
    }

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    public void deposit(double amount) {
        this.balance += amount;
    }

    public void withdraw(double amount) {
        this.balance -= amount;
    }

    protected final double getBalance() {
        return this.balance;
    }
}
```

Subclasses will inherit this field even though they cannot directly access it

- Subclasses inherit *all* fields

Calls the one-parameter constructor

protected means that *subclasses* and classes in the same *package* can access it.

- *public* makes more sense here, but I have made it protected just so that you can see an example

final means that subclasses are not permitted to override this method

- We want to count on it working just like this

```
public class SavingsAccount extends BankAccount {

    private double interestRate;

    public SavingsAccount(double rate) {
        this.interestRate = rate;
    }

    public SavingsAccount(double rate, double initBalance) {
        super(initBalance);
        this.interestRate = rate;
    }

    public void addInterest() {
        double interest;
        interest = this.getBalance()
                        * this.interestRate / 100;
        this.deposit(interest);
    }
}
```

Fields:
- Inherits *balance* field
  - DON'T put your own *balance* field here!
- Adds *interestRate* field

Implicit **super();** that calls superclass' no-parameter constructor

Calls superclass' constructor
- Must be first statement in constructor

Adds this method to those inherited

Calls inherited *getBalance* and *deposit* methods

```
public class CheckingAccount extends BankAccount {
    private int transactionCount;

    public CheckingAccount(double initialBalance) {
        super(initialBalance);
        this.transactionCount = 0;
    }

    @Override
    public void withdraw() {
       super.withdraw();
       ++ this.transactionCount;
    }

    public void runThisOnFirstDayOfMonth) {
        if (this.transactionCount > 100) {
           super.withdraw(10.00);
        }
        this.transactionCount = 0;
    }
}
```

Overrides inherited *withdraw* method and also calls inherited *withdraw* method

- The class would have, but I have not shown, a similar *deposit* method.

This (rather silly) checking account charges a $10 fee if you do more than 100 transactions in a month

- Note call to superclass' *withdraw*

Interfaces vs. Inheritance

- ***class ClickHandler implements MouseListener***
  - ClickHandler promises to implement all the methods of MouseListener

For client code reuse

- ***class CheckingAccount extends BankAccount***
  - CheckingAccount inherits all the fields and methods of BankAccount

For implementation code reuse

Hide implementation, use interface type

- Consider:

```
public void moveTo(Point2D pointToMoveTo)
```

- `Point2D` is an *interface* that includes the methods `getX()` and `getY()`

- `Point2D` has *implementations* that include `Point2D.Double` and `Point` and `Point2D.Float`
  - Your code does not care which implementation it is; it works with any of them.
  - Your code needs only to know that you can get the X and Y components of `pointToMoveTo` by using the methods promised by the `Point2D` interface, e.g. `pointToMoveTo.getX()`

Use Getters and Setters

Bad: locks superclass into using a `Point2D.Double`

In superclass:
```
protected Point2D.Double location;
... this.location = new Point2D.Double(..., ...);
```
In subclass:
```
... this.location.x = this.location.x + ...;
```

In superclass:
```
private Point2D location;
... this.location = new Point2D.Double(..., ...);

protected final Point2D getLocation() {
    return this.location;
}

protected final void setLocation(Point2D location) {
    this.location = location;
}
```

Good: allows superclass to change the implementation of the Ball's location.

Eclipse types most of this code for you!

In subclass:
```
this.setLocation(new Point2D.Double(
                    this.getLocation().getX() + ...,
                    this.getLocation().getY() + ...);
```