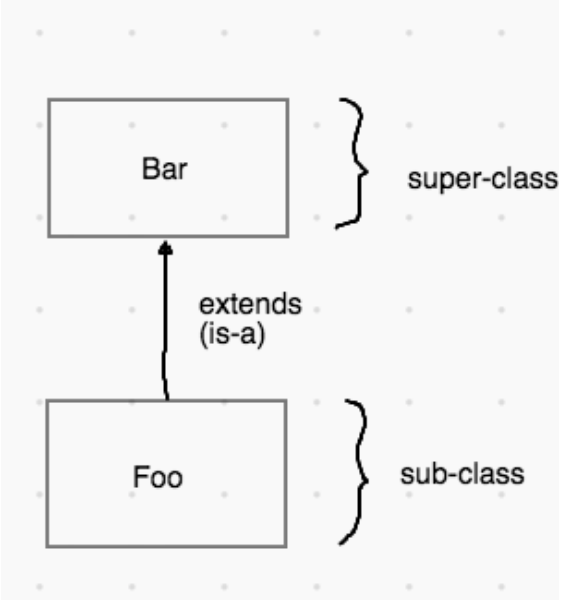
Inheritance

When a class is built on top of another class. Specifically it ”inherits” the original class’ members (fields, constructors, methods, etc…). Described as an ”is-a” relationship, but Java uses the keyword ”extends”:



We call the original class the *super* class and the new, inheriting class, the *sub* class.

How is this useful? There are a few uses, for now we can focus on code reusability: the sub-class does not repeat the members of the super-class.

Example: Modeling Bank Accounts

Representing a basic bank account

What would you need to store an account? account number, balance, SIN (or other personal info), name, phone, type(!)

What operations would you support? withdraw, deposit, getters

```
public class Account {

    private long accountId;
```

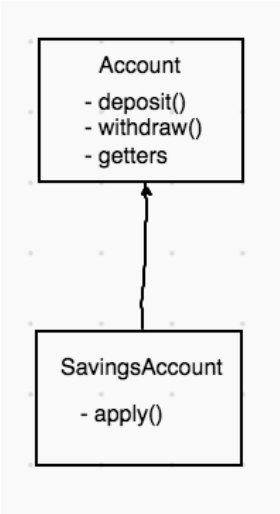
```
    private String holder;
    private double balance;

    public Account(long accountId, String holder, double balance) {
        this.accountId = accountId;
        this.holder = holder;
        this.balance = balance;
    }

    public long getAccountId() {
        return accountId;
    }

    public String getHolder() {
        return holder;
    }

    public double getBalance() {
        return balance;
    }

    public void withdraw(double amt) {
        if(amt < 0 || amt > balance)
            throw new IllegalArgumentException();
        balance -= amt;
    }

    public void deposit(double amt) {
        if(amt < 0)
            throw new IllegalArgumentException();
        balance += amt;
    }
}
```

Here's an example of how we can create an account:

```
Account account = new Account(123l, "Ian", 100.0);
account.deposit(1000.0);
account.withdraw(50.0);
System.out.println(account.getBalance());
```

Bank account types: savings account

Extends a basic account with the idea that interest can be earned on the balance. There is now an interest rate and an operation `apply`. We will inherit the `Account` class members and add this new feature:




```
public class SavingsAccount extends Account {

    private double rate;

    public SavingsAccount(long accountId, String holder, double balance, double rate) {
        super(accountId, holder, balance);
        this.rate = rate;
    }

    public void apply() {
        //balance += balance * rate;
        deposit(getBalance() * rate);
    }

}
```

Here's the example of how we can create a savings account:

```
SavingsAccount savingsAccount = new SavingsAccount(124l, "Sandy", 100.0, 0.02);
savingsAccount.deposit(1000.0);
```

```
savingsAccount.withdraw(50.0);
savingsAccount.apply();
System.out.println(savingsAccount.getBalance());
```

Notice:

1. We use the keyword `extends` to indicate the super-class.
2. We need to write a constructor that initializes the super-class members, i.e.: calls the super-class constructors. This usually, though not always, means that we need to write a sub-class constructor that includes the parameters of the super-class constructor. We call the super-class constructor from the sub-class constructor using the `super( .. )` call.
3. Super-class members are available in the sub-class and to the other parts of the program.
4. Private super-class members are private to the sub-class. They are still part of the sub-class, since they form the overall object but they are hidden or invisible to the sub-class code.

Exercise: Extending the Stack

Create a class called `StackMany` that supports the original stack operations with two new ones:

push-many

Description	Push multple elements onto the stack in LIFO order
Signature	`void pushMany(T[] elements)`
Preconditions	Stack has room for the number of elements
Returns	None

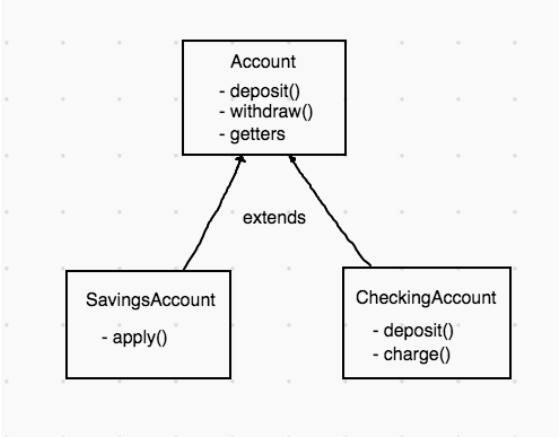
pop-many

Description	Pop multiple elements from the stack in LIFO order
Signature	`T[] popMany(int amount)`
Preconditions	Stack has the number of elements, amount > 0
Returns	the popped elements in LIFO order

Example: Modeling Bank Accounts (continued)

Bank account types: checking account

Extends a basic account with the idea that transactions (withdrawals only) can be charged. There is now an withdrawal counter and an operation `charge`. We will inherit the `Account` class members and add this new feature:



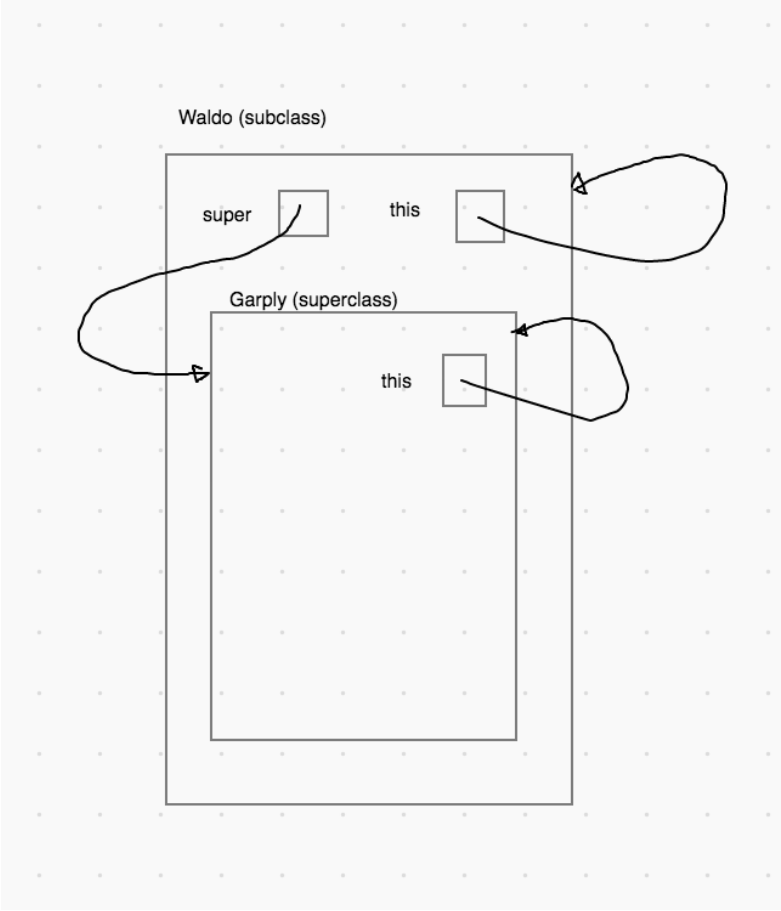

```
public class CheckingAccount extends Account {

    public static final double CHARGE_PER_WITHDRAW = 0.25;
    private int withdrawCount;

    public CheckingAccount(long accountId, String holder, double balance) {
        super(accountId, holder, balance);
        withdrawCount = 0;  // unnecessary since integers default to 0
    }

    @Override
    public void withdraw(double amt) {
        super.withdraw(amt);
        withdrawCount++;
    }

    public void charge() {
        withdraw(withdrawCount * CHARGE_PER_WITHDRAW);
        withdrawCount = 0;
    }
}
```

Here's the example of how we can create a checking account:

```
CheckingAccount checkingAccount = new CheckingAccount(125l, "Helen", 100.0);
checkingAccount.deposit(1000.0);
checkingAccount.withdraw(50.0);
checkingAccount.charge();
System.out.println(checkingAccount.getBalance());
```

Notice:

1. We can redifine a method, called a method *override*.
2. The original super-class method is still called when the object is the super-class, and the new version is called with the object is the sub-class.
3. Inside the sub-class, you can still make use of the super-class method by using the `super` keyword.



Exercise: Extending the Queue

Create a class `QueueDefault` that supports the original queue operations with the following update: dequeue will either return an enqueued value, or if there are none, will return the default value. The queue never appears as empty.

Three type of methods

1. Methods that are only members of the superclass.
2. Methods that are only members of the subclass.
3. Methods that are members of the superclass that are redefined in the subclass, called a method override. Java uses the method signatures to figure out the override, i.e.: the method name and the parameters (datatypes) must match. The `@Override` annotation is optional, but very helpful.

   There are two kinds of overrides:

   a) overrides that completely replace the superclass version. b) overrides that makes use of the superclass version. These will call the super-class version using the `super` keyword.

Example:

Here's an example of how these methods are accessed:

```
public class Bar {

    public void a() {
        System.out.println("Bar.a");
    }

    public void c() {
        System.out.println("Bar.c");
    }

    public void d() {
        System.out.println("Bar.d");
    }
}
```

```
public class Foo extends Bar {

    public void b() {
        System.out.println("Foo.b");
    }

    @Override
    public void c() {
        System.out.println("Foo.c");
    }

    public void d() {
        super.d();
        System.out.println("Foo.d");
    }
}
```

When we instantiate these classes and call each method we get the this:

```
Bar bar = new Bar();
bar.a(); // -> Bar.a();
// bar.b(); -> compiler error
bar.c(); // -> Bar.c();
bar.d(); // -> Bar.d();

Foo foo = new Foo();
foo.a(); // -> Bar.a()
foo.b(); // -> Foo.b()
foo.c(); // -> Foo.c();
foo.d(); // -> Bar.d() then Foo.d()
```

Exercise: Designing using inheritance

Restructure (aka. refactor) the survey example. Replacing the `Questionable` interface with a `Question` superclass.

Advice:

- Look to see what's common among the current questions' methods and what is done differently in each.
- Store as much info as you can in the `Question` class.

- Think about what behaviours will be overridden by more complicated question styles.
- There are two solutions here: one that will include an `abstract` method in your design and one that doesn't. Either are good!

Videos:

- `https://youtu.be/8LPjwdcQQfY`
- `https://youtu.be/kQy8dHOYkjI`
- `https://youtu.be/wSNQ4p4HI6A`
- `https://youtu.be/toWa_rIYGq0`
- `https://youtu.be/x1YdJPggaqw`
- coming soon