Abstract and Interfaces

Abstract Classes and Methods

Interfaces

Polymorphism with Interfaces

Abstract Classes and Methods

?

Weapon
Should this exist as an object?

Practice Sword
Very weak (2 hits to kill the weakest enemy)

Goddess Sword
Stronger, and can be altered for new capabilities

?

Weapon
Should this exist as an object?

Concrete objects

Practice Sword
Very weak (2 hits to kill the weakest enemy)

Goddess Sword
Stronger, and can be altered for new capabilities

?

Weapon
Should this exist as an object?

Abstract object

Practice Sword
Very weak (2 hits to kill the weakest enemy)

Goddess Sword
Stronger, and can be altered for new capabilities

Concrete Classes

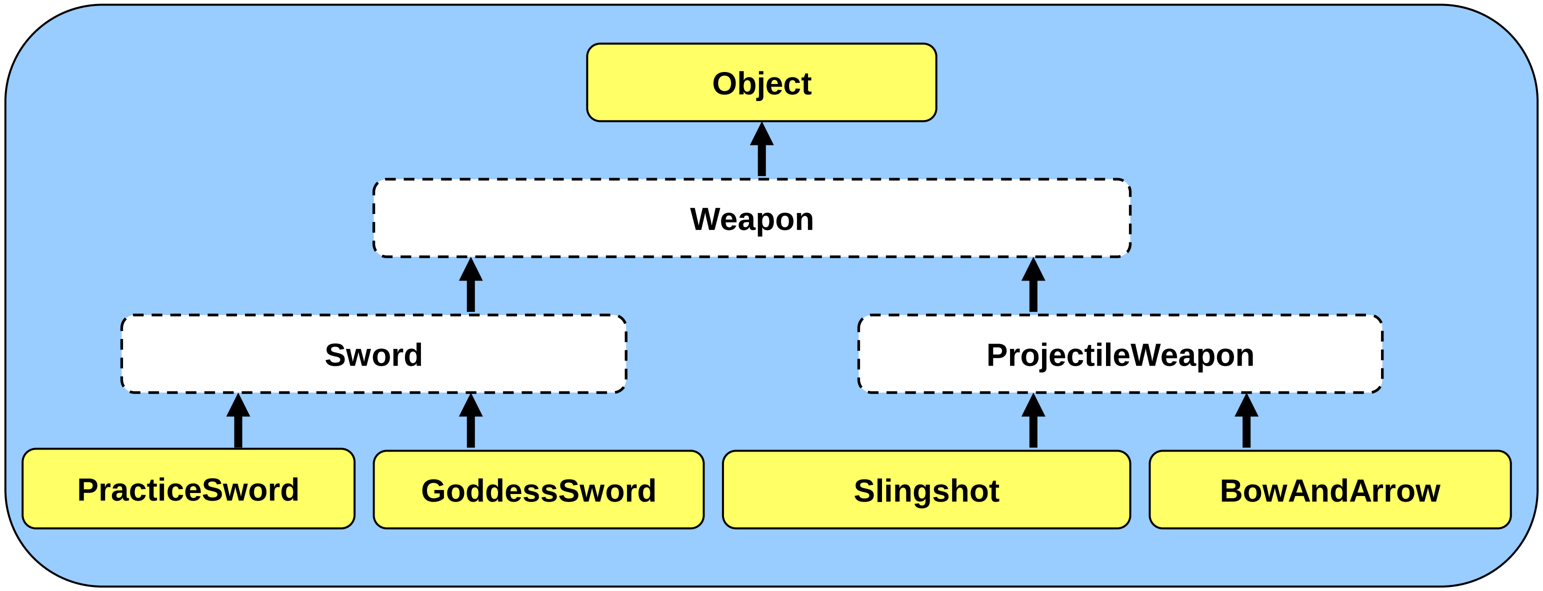
classes that we can make instances of directly by using the `new` keyword

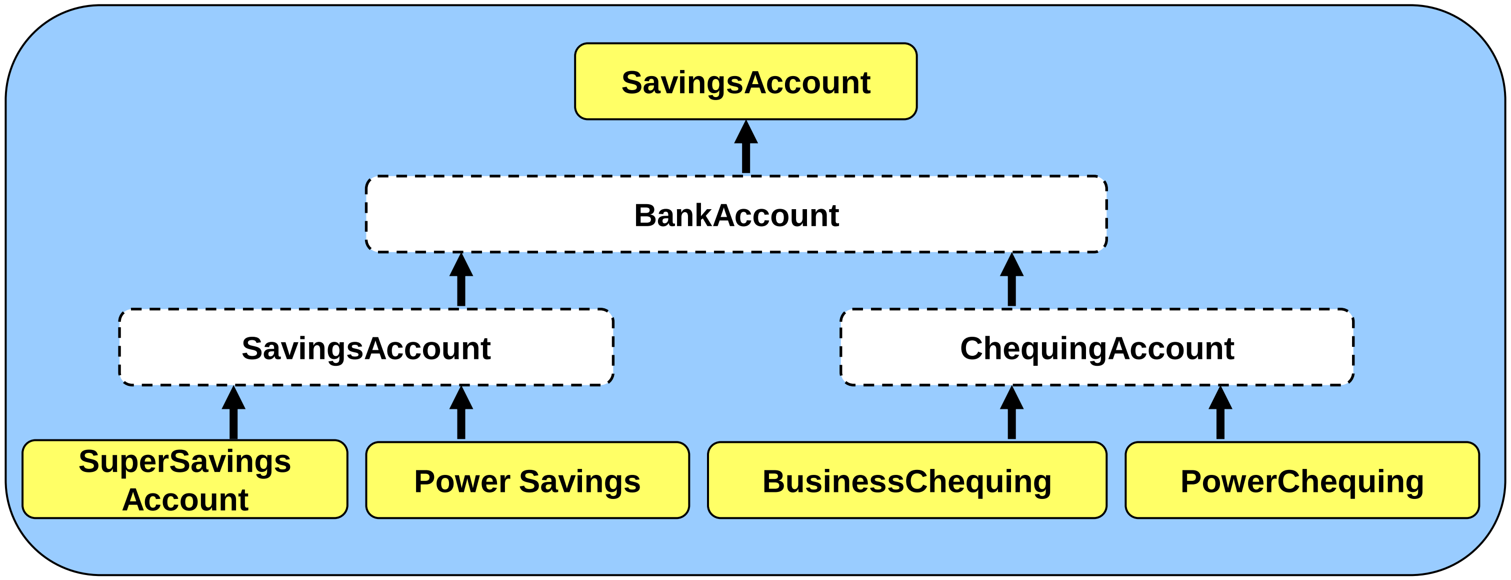
Abstract Classes

classes for which we cannot create instances

Abstract Methods

methods with no code for which
all concrete subclasses are forced
to implement the method


Object
Weapon
Sword
ProjectileWeapon
PracticeSword
GoddessSword
Slingshot
BowAndArrow


SavingsAccount
BankAccount
SavingsAccount
ChequingAccount
SuperSavings Account
Power Savings
BusinessChequing
PowerChequing

```
public abstract class BankAccount
{
    public abstract void deposit();
    ...
}

public abstract class SavingsAccount
    extends BankAccount
{
    ...
}

public abstract class ChequingAccount
    extends BankAccount
{
    ...
}
```

```
account1 = new SuperSavings(…);
account2 = new PowerSavings(…);
account3 = new BusinessChequing(…);
account4 = new PowerChequing(…);

new BankAccount(…)      // does not compile
new SavingsAccount(…)   // does not compile
new ChequingAccount(…) // does not compile
```

Specific bank account types

```
account1 = new SuperSavings(…);
account2 = new PowerSavings(…);
account3 = new BusinessChequing(…);
account4 = new PowerChequing(…);
```

```
new BankAccount(…)     // does not compile
new SavingsAccount(…)  // does not compile
new ChequingAccount(…) // does not compile
```

```
account1 = new SuperSavings(…);
account2 = new PowerSavings(…);
account3 = new BusinessChequing(…);
account4 = new PowerChequing(…);
```

```
new BankAccount(…)      // does not compile
new SavingsAccount(…)   // does not compile
new ChequingAccount(…)  // does not compile
```

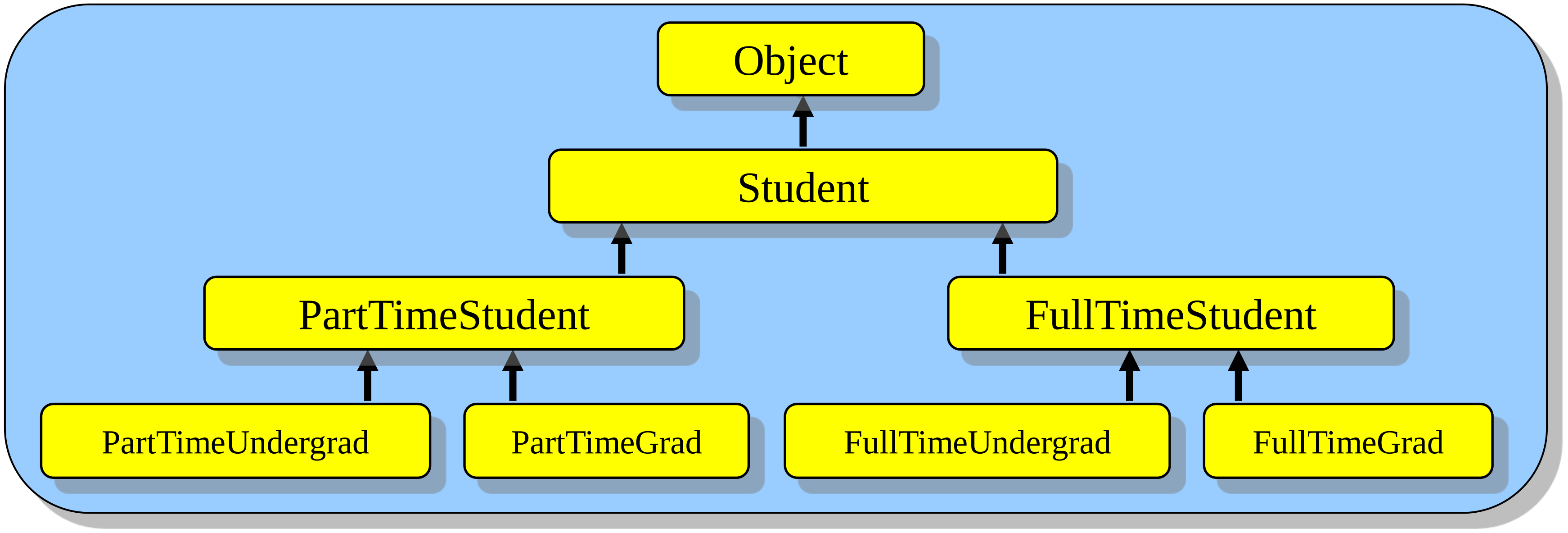
Not specific enough – a teller would ask what *kind* of bank account before opening one

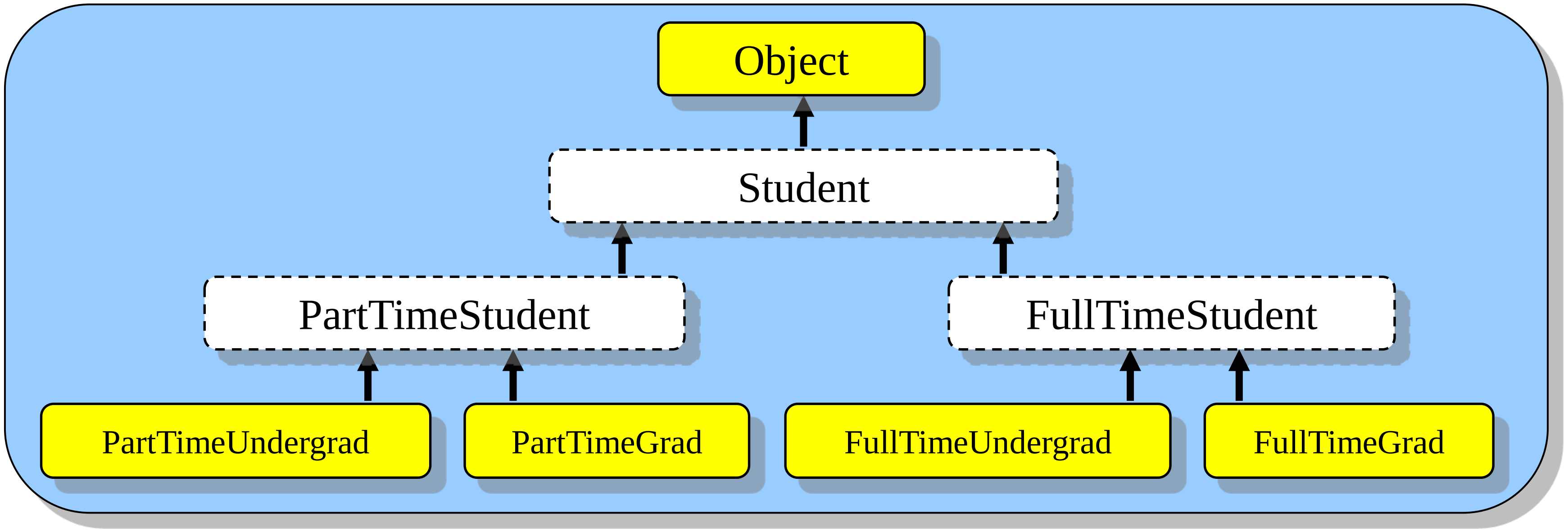
Why make classes abstract?

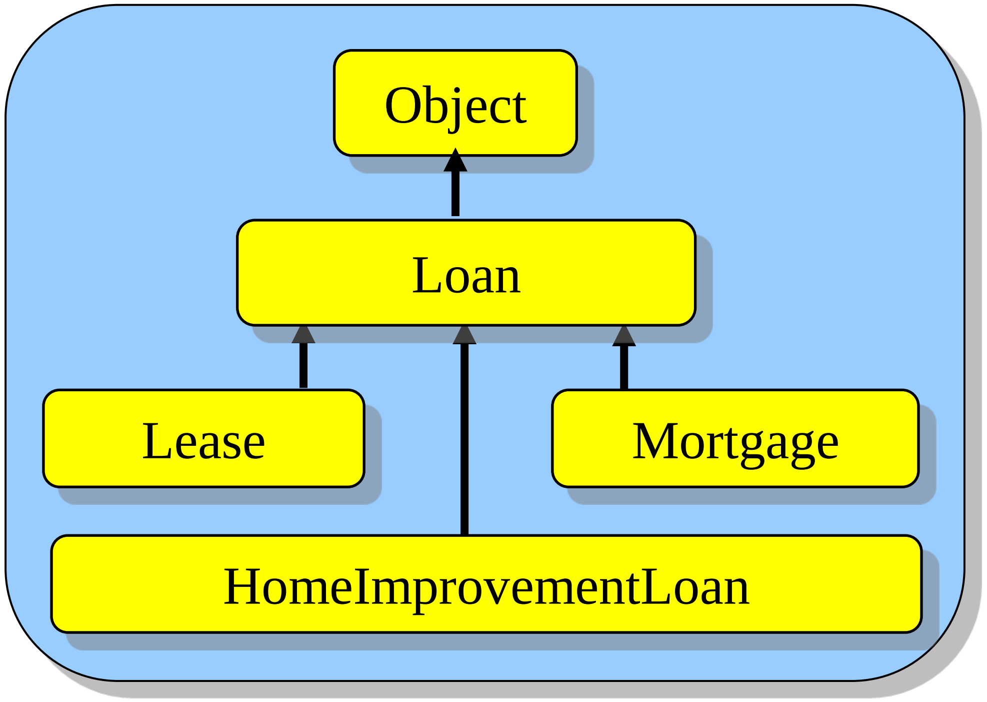
Get advantages of inheritance while informing users of class's purpose and forcing them to subclass your class.

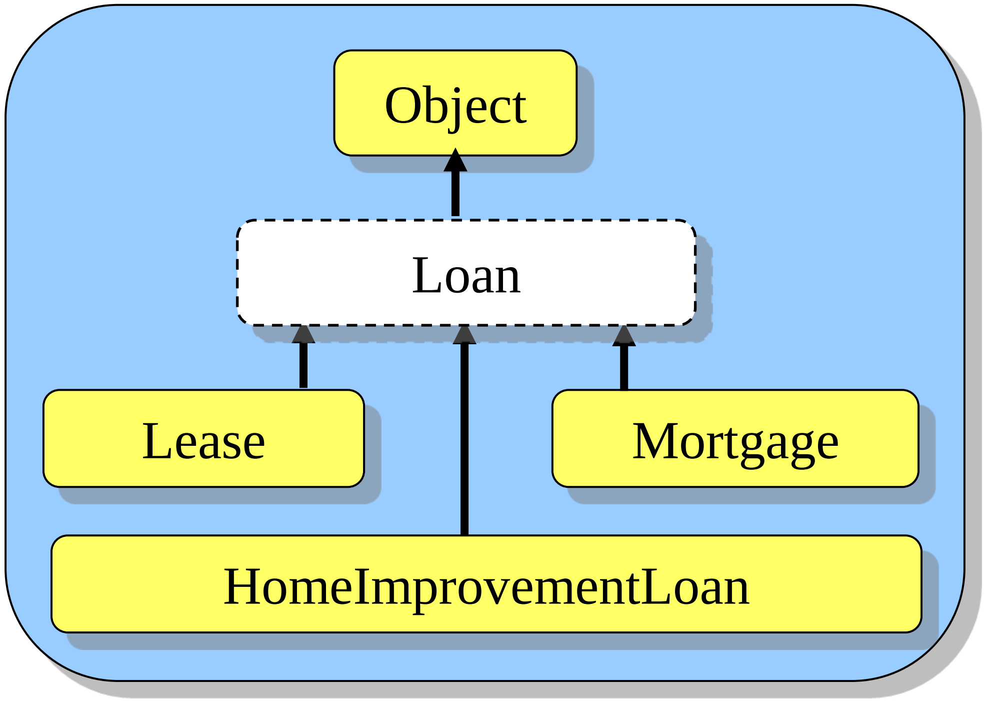
Which classes should we make abstract?

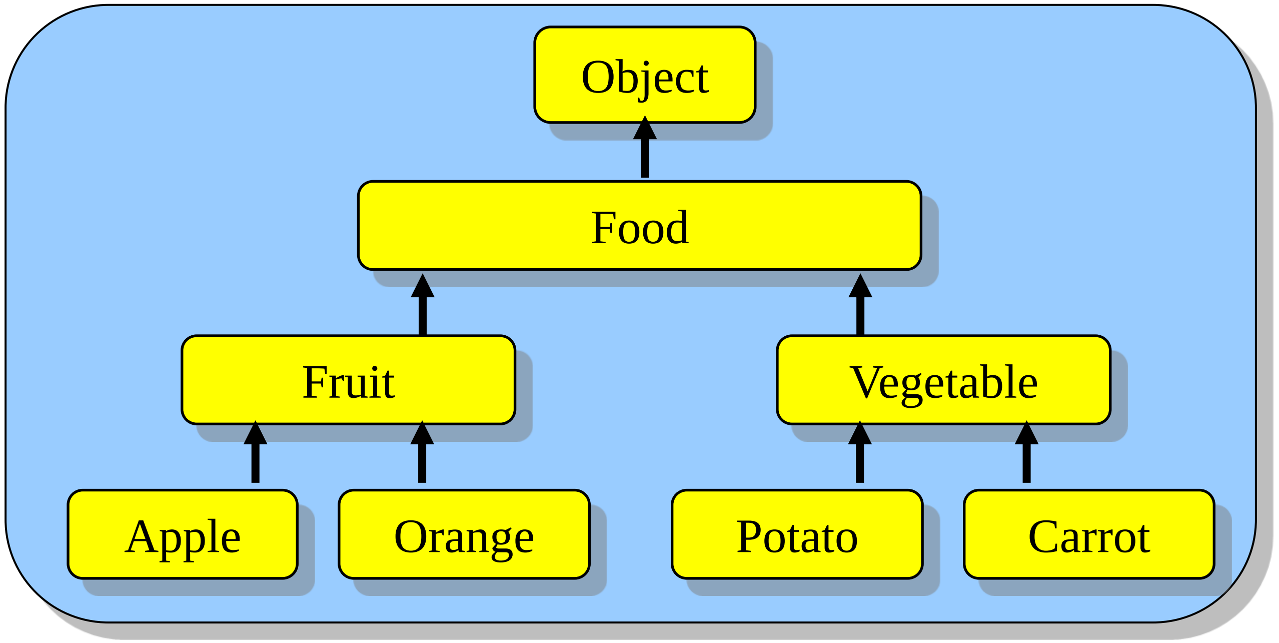
If not sure, leave a class concrete. Otherwise, if all subclasses cover the possible concrete classes that will be needed, make the superclass abstract.

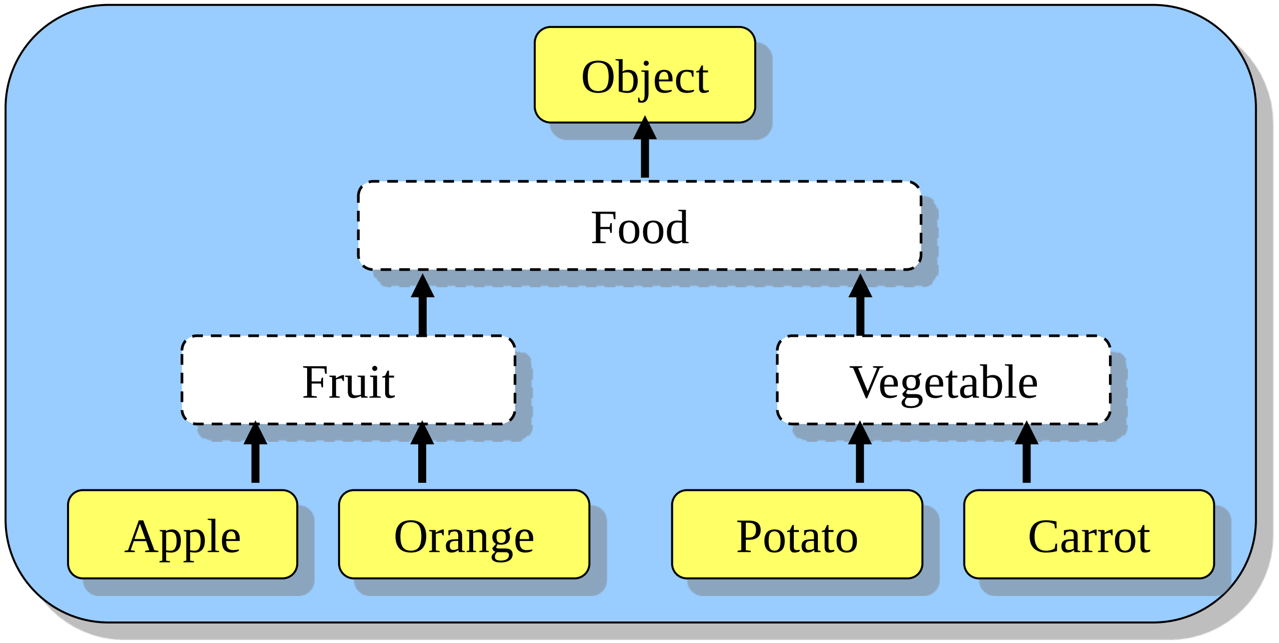

Object
Student
PartTimeStudent
FullTimeStudent
PartTimeUndergrad
PartTimeGrad
FullTimeUndergrad
FullTimeGrad


Object
Student
PartTimeStudent
FullTimeStudent
PartTimeUndergrad
PartTimeGrad
FullTimeUndergrad
FullTimeGrad


Object
Loan
Lease
Mortgage
HomeImprovementLoan


Object
Loan
Lease
Mortgage
HomeImprovementLoan


Object
Food
Fruit
Vegetable
Apple
Orange
Potato
Carrot


Object
Food
Fruit
Vegetable
Apple
Orange
Potato
Carrot

Poll Everywhere Question

What will the result be of compiling and running the following code?

```
abstract class Employee {
    String name;
    public abstract float calcIncome();
}

class Manager extends Employee {
    public void hire(String who) {
        System.out.println( who + " hired by "
                            + name );
    }

    public void fire(String who) {
        System.out.println( who + " fired by "
                            + name );
    }
}

public class ManagerCheck {
    public static void main(String args[]) {
        Manager me = new Manager();
        me.hire("newbie");
        me.fire("nobody");
    }
}
```

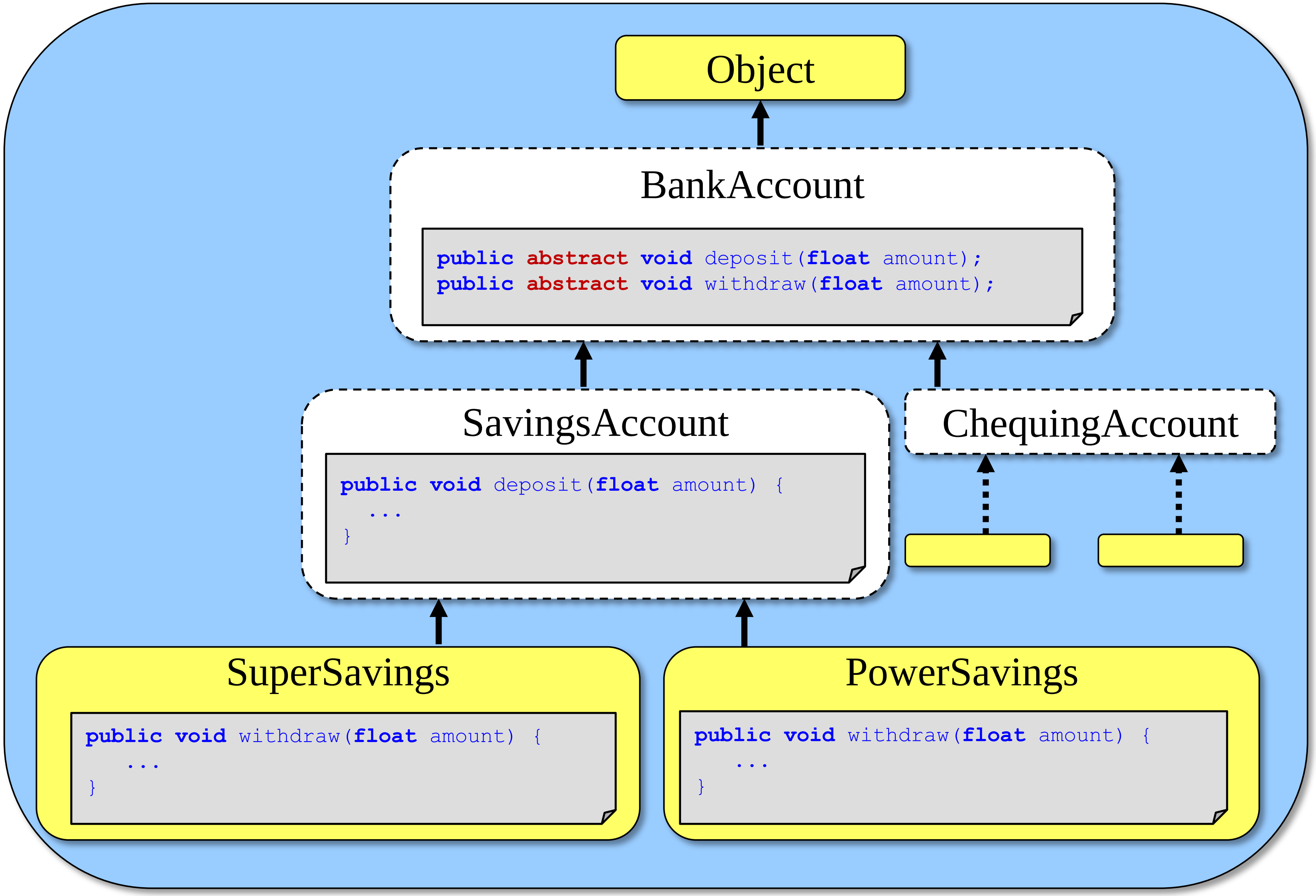
Text 37607

68936:
```
who hired by name
who fired by name
```

68938:
```
newbie hired by name
nobody fired by name
```

68953:
```
newbie hired by null
nobody fired by null
```

69082:
Compilation fails



```
public abstract void deposit(float amount);
public abstract void withdraw(float amount);
```

```
public void deposit(float amount) {
   ...
}
```

```
public void withdraw(float amount) {
   ...
}
```

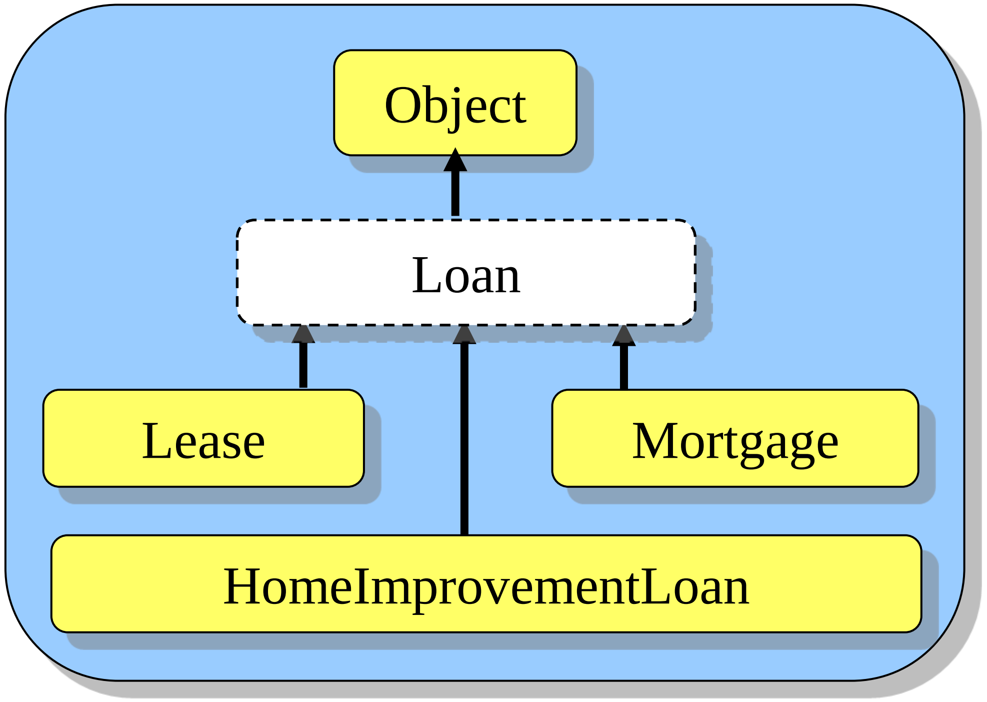
```
public void withdraw(float amount) {
   ...
}
```

When should methods be abstract?

When you want to force concrete subclasses to write their own custom version of the behaviour.

When should methods *not* be abstract?

When there is a default behaviour, just write the code in the abstract class.



All loans:
Get client information
etc

Only some types:
Refinancing
etc

```
public abstract class Loan
{
     public abstract float calculateMonthlyPayment();
     public abstract void makePayment(float amount);
     public abstract void renew(int numMonths);

     public Client getClientInfo()
     {
          ...
     }

     ....
}
```

Abstract: subclasses must implement

```
public abstract class Loan
{
    public abstract float calculateMonthlyPayment();
    public abstract void makePayment(float amount);
    public abstract void renew(int numMonths);

    public Client getClientInfo()
    {
        ...
    }

    ....
}
```

```
public abstract class Loan
{
    public abstract float calculateMonthlyPayment();
    public abstract void makePayment(float amount);
    public abstract void renew(int numMonths);

    public Client getClientInfo()
    {
        ...
    }

    ....
}
```

Non-abstract: no need to override it

```
public abstract class Loan
{
    public float  calculateMonthlyPayment(){ return 0;}
    public void   makePayment(float amount){ }
    public void   renew(int numMonths){ }
    public Client getClientInfo() { ... }
    ....
}
```

What if no methods were abstract?

```
public abstract class Loan
{
    public float  calculateMonthlyPayment(){ return 0;}
    public void   makePayment(float amount){ }
    public void   renew(int numMonths){ }
    public Client getClientInfo() { .... }
    ....
}
```

These need bodies, even if they're empty

```
public abstract class Loan
{
    public float  calculateMonthlyPayment(){ return 0;}
    public void   makePayment(float amount){ }
    public void   renew(int numMonths){ }
    public Client getClient
    ....
}
```

Subclasses aren't required to override them anymore

Interfaces

Interface

a specification (i.e., a list) of a set of methods such that any classes implementing the interface are forced to write

Interface

a specification (i.e., a list) of a set of methods such that any classes implementing the interface are forced to write

Much like abstract methods

```
public interface Loanable
{
    public float calculateMonthlyPayment();
    public void  makePayment(float amount);
    public void  renew(int numMonths);
}
```

Instead of `class`

```
public interface Loanable
{
    public float calculateMonthlyPayment();
    public void  makePayment(float amount);
    public void  renew(int numMonths);
}
```

Defined in its own Java file

```
public interface Loanable
{
    public float calculateMonthlyPayment();
    public void  makePayment(float amount);
    public void  renew(int numMonths);
}
```

```
public interface Loanable
{
    public float calculateMonthlyPayment();
    public void  makePayment(float amount);
    public void  renew(int numMonths);
}
```

Methods can never have a body

Methods must always be public

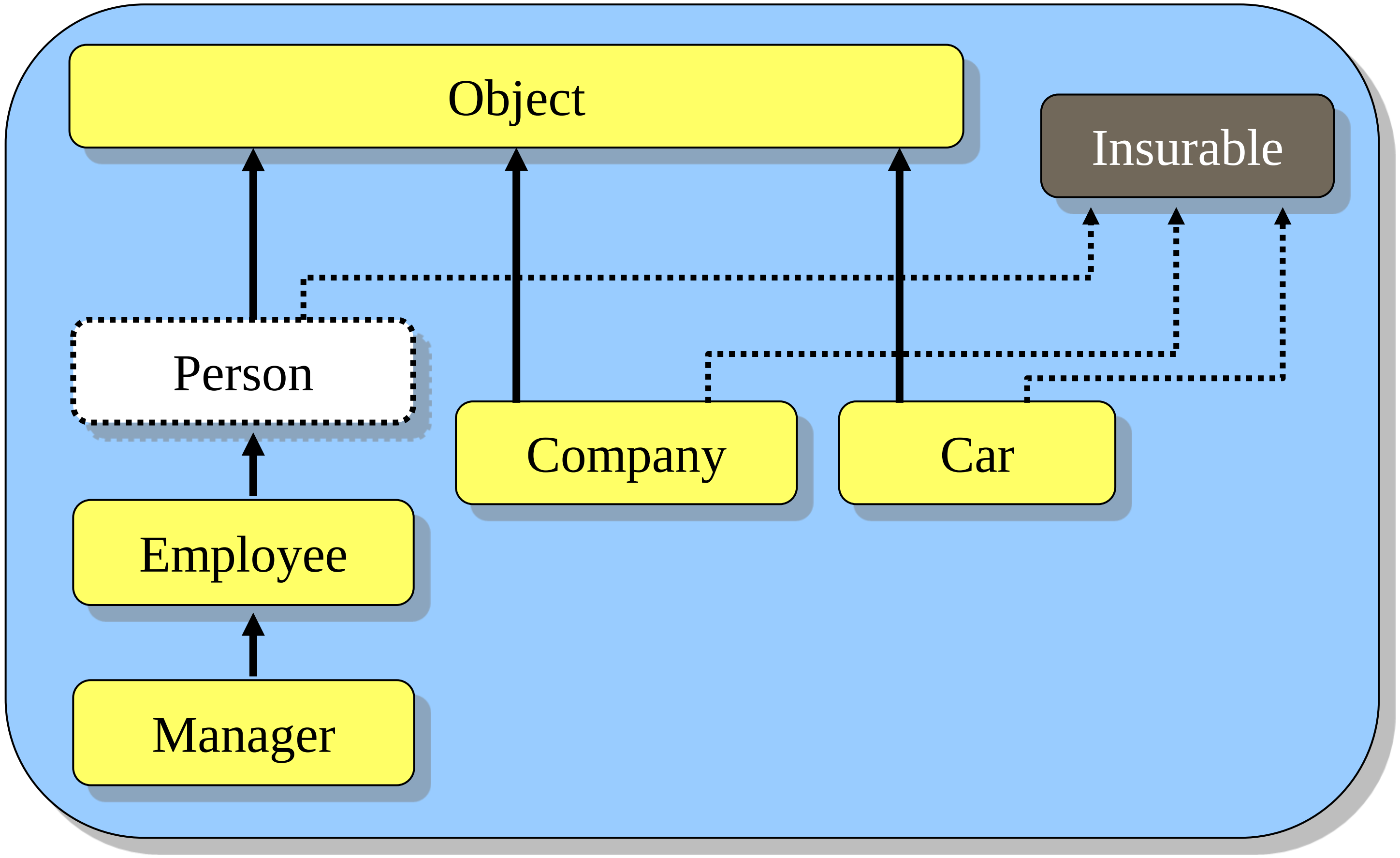
```
p                         nable
{
    public float calculateMonthlyPayment();
    public void  makePayment(float amount);
    public void  renew(int numMonths);
}
```

```
public interface Loanable
{
    public float calculateMonthlyPayment();
    public void  makePayment(float amount);
    public void  renew(int numMonths);
}
```

Can't make a new instance: `new Loanable()` is an error

```java
public interface Loanable
{
    public float calculateMonthlyPayment();
    public void  makePayment(float amount);
    public void  renew(int numMonths);
}
```

Can’t have any attributes or static constants


Object
Insurable
Person
Company
Car
Employee
Manager

```
public interface Insurable
{
    public int getPolicyNumber();
    public int getCoverageAmount();
    public double calculatePremium(int days);
    public java.util.Date getExpiryDate();
}
```

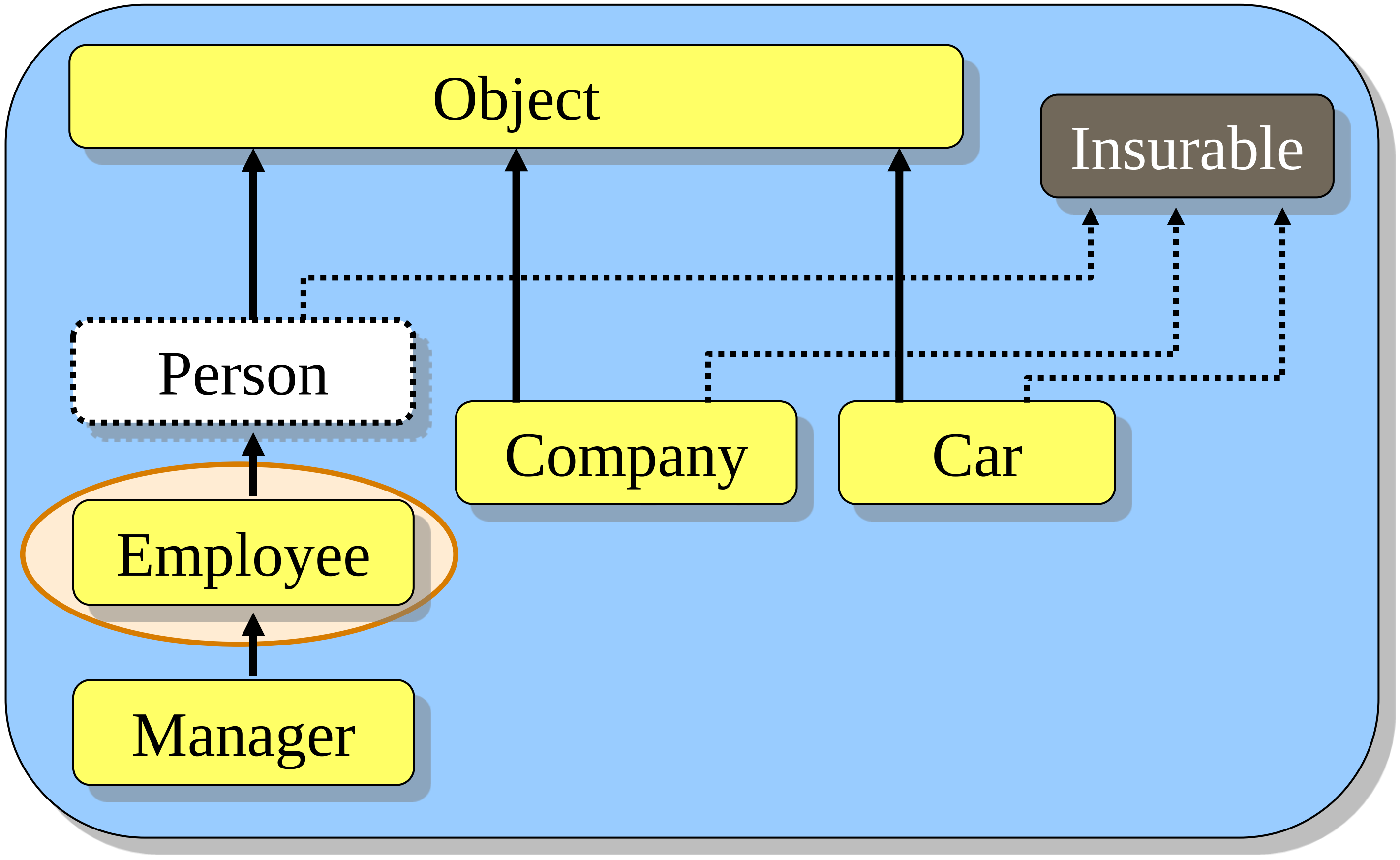
```
public class Person implements Insurable
{
    ...
}

public class Company implements Insurable
{
    ...
}

public class Car implements Insurable
{
    ...
}
```

```
public class Person implements Insurable
{
    ...
}

public class                           Insurable
{
    ...
}

public class Car implements Insurable
{
    ...
}
```

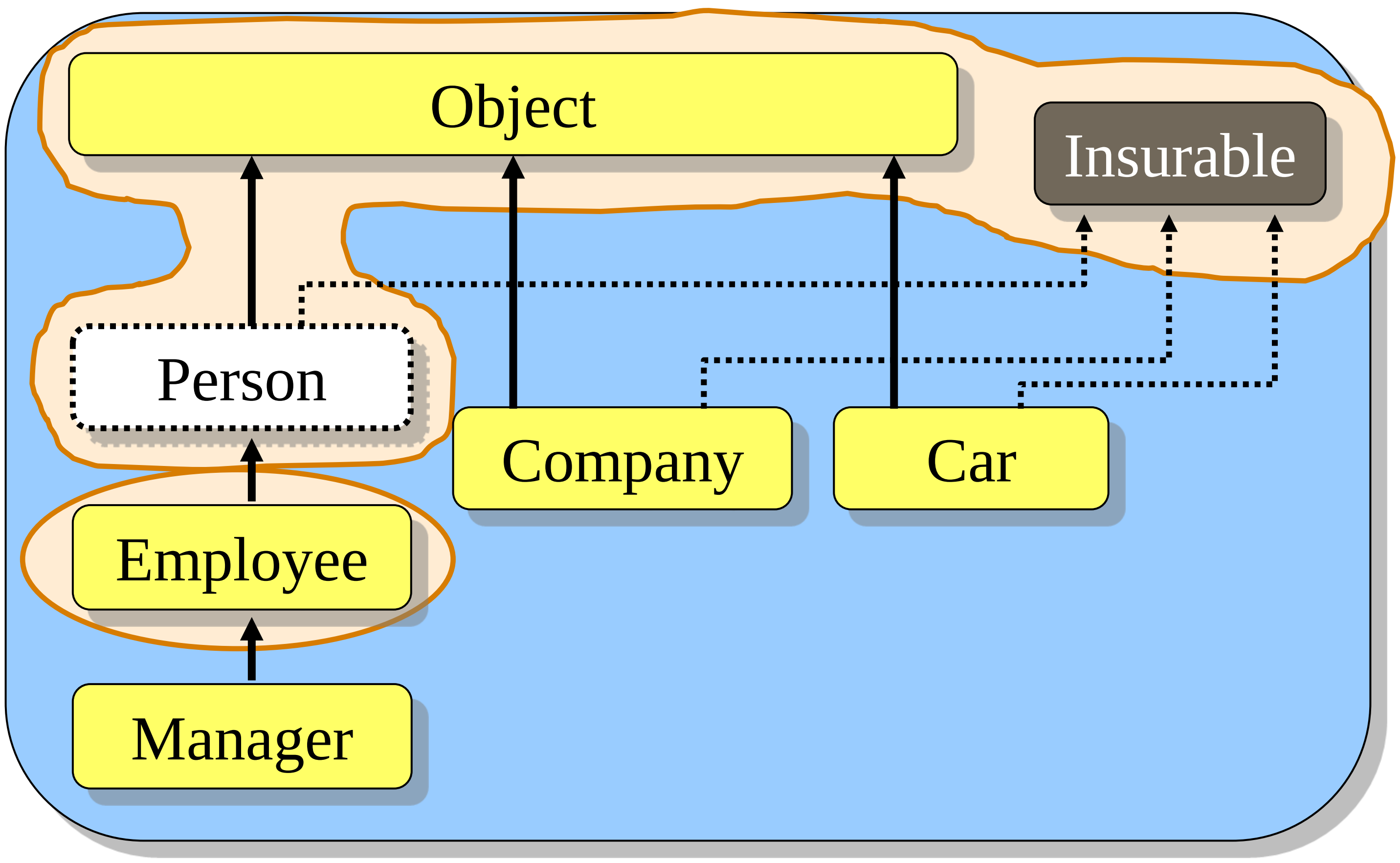
Each class has to implement all methods from the interface

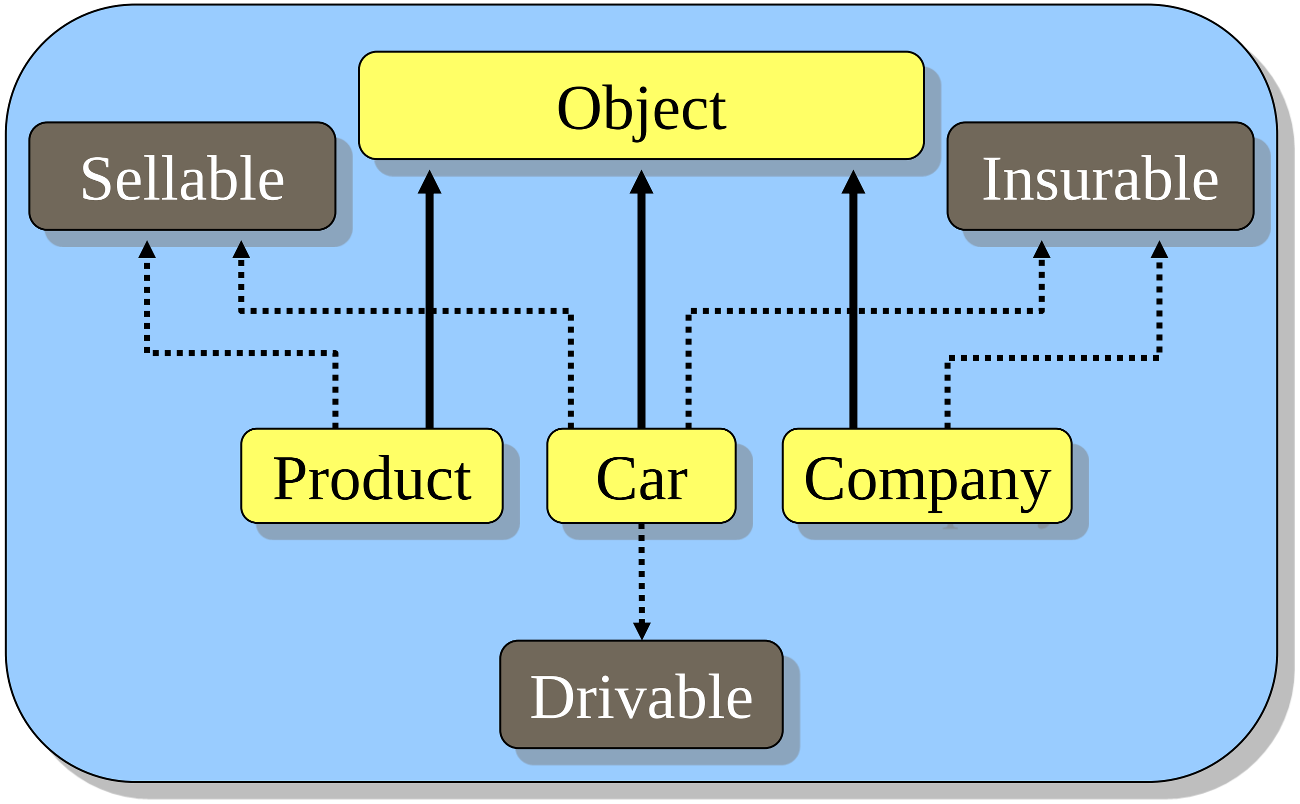
```
public class Person implements Insurable
{
    ...
}

public class Company implements Insurable
{
    ...
}

public class Car implements Insurable
{
    ...
}
```

APPROVED

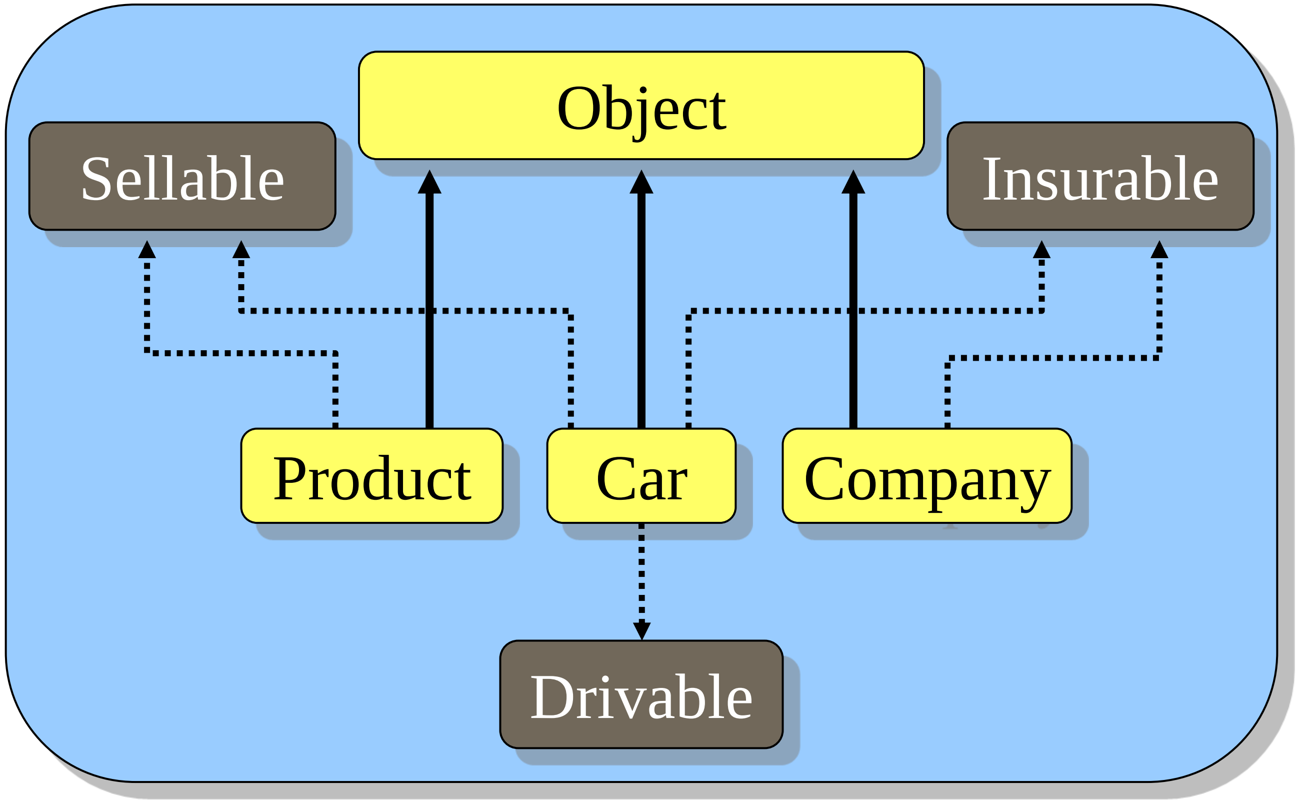


What can an `Employee` be cast to?

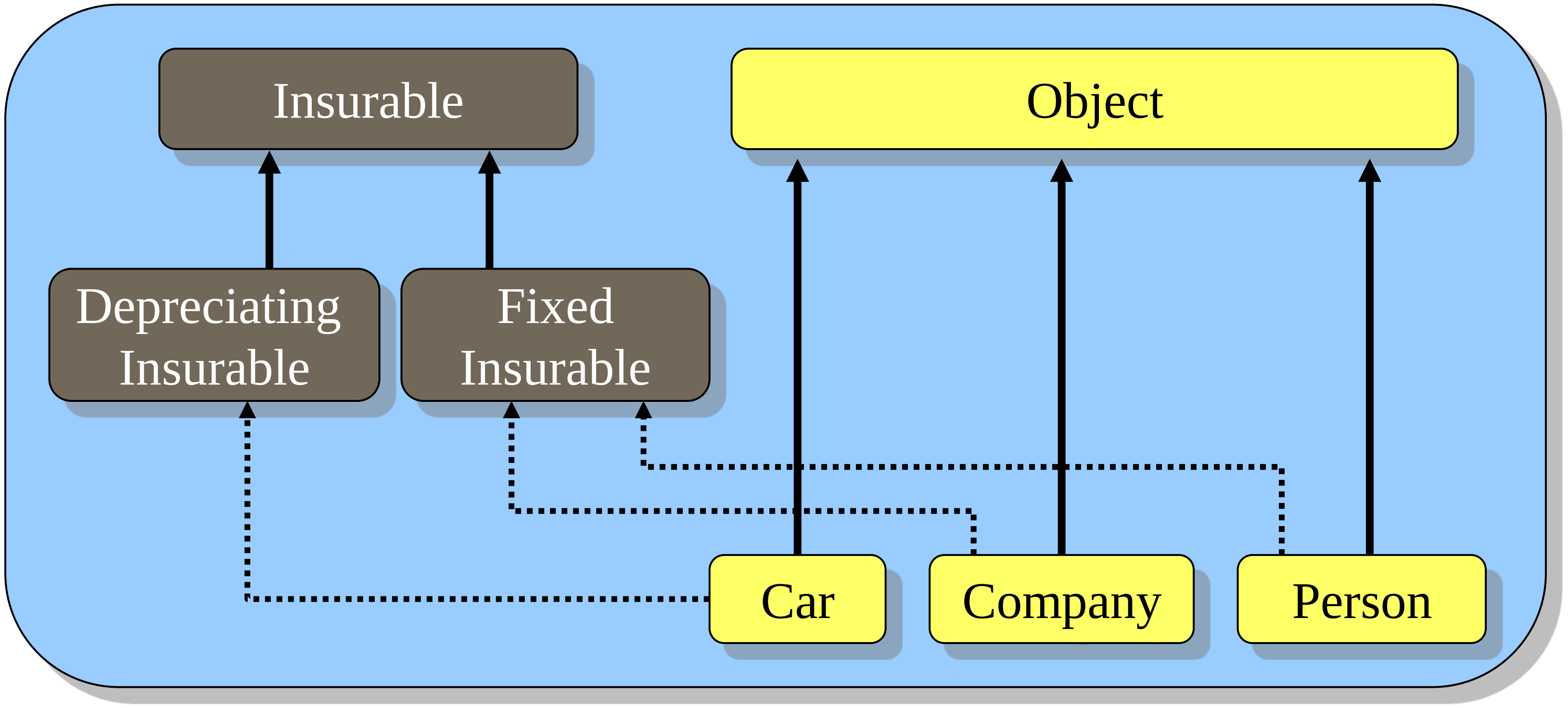

Object
Insurable
Person
Company
Car
Employee
Manager



An object can implement multiple interfaces



```
public class Car implements Insurable, Drivable, Sellable
{
    ...
}
```



Interfaces can inherit from each other just like classes

```
public interface Insurable
{
    public int getPolicyNumber();
    public int getCoverageAmount();
    public double calculatePremium(int days);
    public java.util.Date getExpiryDate();
}
```

```
public interface DepreciatingInsurable extends Insurable
{
    public double computeFairMarketValue();
    public void amortizePayments();
}
```

```
public interface Insurable
{
    public int getPolicyNumber();
    public int getCoverageAmount();
    public doubl
    public java.
}
```

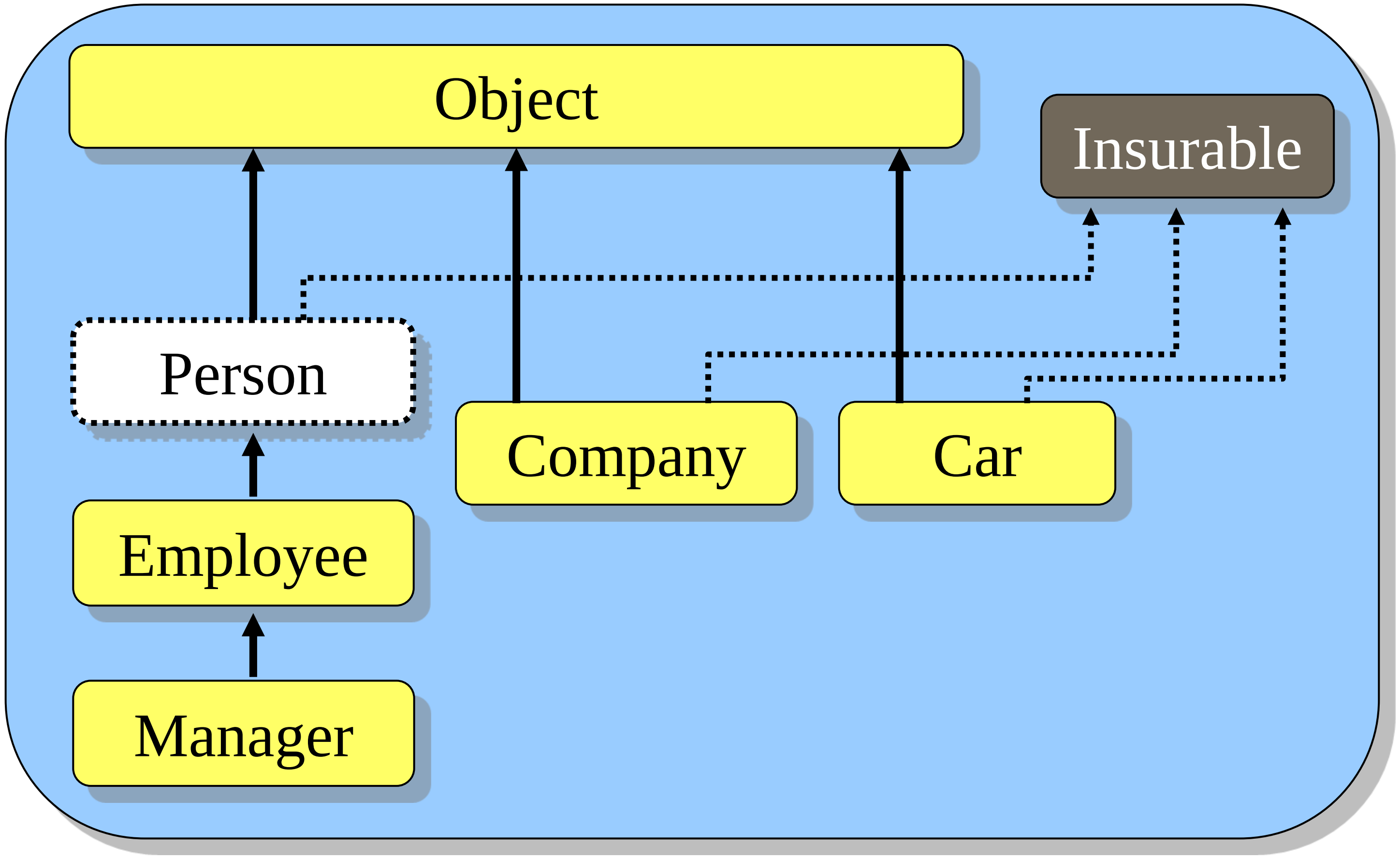
Classes that implement `DepreciatingInsurable` **must** implement `Insurable` **methods too**

```
public interface DepreciatingInsurable extends Insurable
{
    public double computeFairMarketValue();
    public void amortizePayments();
}
```

Why use interfaces?

They allow us to specify common behavior between otherwise unrelated objects.

Polymorphism with Interfaces


Object
Insurable
Person
Company
Car
Employee
Manager

```
Car         jetta = new Car();
Insurable   item = (Insurable)jetta;

item.getPolicyNumber();
jetta.getMileage();
item.getMileage();
((Car)item).getMileage();
```

```
Car          jetta = new Car();
Insurable    item = (Insurable)jetta;

item.getPolicyNumber();
jetta.getMileage();
item.getMileage();
((Car)item).getMileage();
```

Ok since `Insurable`

```
Car          jetta = new Car();
Insurable    item = (Insurable)jetta;

item.getPolicyNumber(
jetta.getMileage();
item.getMileage();
((Car)item).getMileage();
```

Ok, assuming this is a car method

```
Car          jetta = new Car();
Insurable    item = (Insurable)jetta;

item.getPolicyNumber();
jetta.getMileage();
item.getMileage();
((Car)item).getMileage();
```

Compile error

```
Car          jetta = new Car();
Insurable    item = (Insurable)jetta;

item.getPolicyNumber();
jetta.getMileage();
item.getMileage();
((Car)item).getMileage();
```

Ok thanks to casting

```
float         total = 0;
Insurable[]   insurableItems;

insurableItems = new Insurable[5];
insurableItems[0] = new Car("Porshce", "Carerra", "Red", 340);
insurableItems[1] = new Customer("Guy Rich");
insurableItems[2] = new Company("Elmo’s Edibles", 2009);
insurableItems[3] = new Employee("Jim Socks");
insurableItems[4] = new Manager("Tim Burr");

System.out.println("Here are the policies:");
for (int i=0; i<insurableItems.length; i++)
{
    System.out.println("  " +
                       insurableItems[i].getPolicyNumber());

    total += insurableItems[i].getPolicyAmount();
}

System.out.println("Total policies amount is $" + total);
```

A single collection of objects that implement `Insurable`

```
float        total = 0;
Insurable[]  insurableItems;

insurableItems = new Insurable
insurableItems[0] = new Car("Porshce", "Carerra", "Red", 340);
insurableItems[1] = new Customer("Guy Rich");
insurableItems[2] = new Company("Elmo’s Edibles", 2009);
insurableItems[3] = new Employee("Jim Socks");
insurableItems[4] = new Manager("Tim Burr");

System.out.println("Here are the policies:");
for (int i=0; i<insurableItems.length; i++)
{
    System.out.println("  " +
                       insurableItems[i].getPolicyNumber());

    total += insurableItems[i].getPolicyAmount();
}

System.out.println("Total policies amount is $" + total);
```

```
float           total = 0;
Insurable[]     insurableItems;

insurableItems = new Insurable[5];
insurableItems[0] = new Car("Porshce", "Carerra", "Red", 340);
insurableItems[1] = new Customer("Guy Rich");
insurableItems[2] = new Company("Elmo's Edibles", 2009);
insurableItems[3] = new Employee("Jim Socks");
insurableItems[4] = new Manager("Tim Burr");

System.out.println("Here are the policies:");
for (int i=0; i<insurableItems.length; i++)
{
    System.out.println("  " +
                        insurableItems[i].getPolicyNumber());

    total += insurableItems[i].getPolicyAmount();
}

System.out.println("Total policies amount is $" + total);
```

Each object is implicitly cast to `Insurable`

```
float          total = 0;
Insurable[]    insurableItems;

insurableItems = new Insurable[5];
insurableItems[0] = new Car("Porsh                          );
insurableItems[1] = new Customer("
insurableItems[2] = new Company("F
insurableItems[3] = new Employee("
insurableItems[4] = new Manager("T

System.out.println("Here are the p
for (int i=0; i<insurableItems.len
{
    System.out.println("   " +
                        insurableItems[i].getPolicyNumber());

    total += insurableItems[i].getPolicyAmount();
}

System.out.println("Total policies amount is $" + total);
```

Notice we don't have to cast back to `Car`, `Customer`, etc, yet the right method version is called