CS 211
INHERITANCE

Inheritance

Example

```
public class Person {
        public String name;
        public int age;
}
```

```
public class Student extends Person{
        public int studentID;
}
```

- A Person object has 2 instance variables: **name**, **age**
- A Student object has 3 instance variables: **name**, **age**, **studentID**
- the Student class is a **subclass** of the Person class.

What is Inheritance?

- **Inheritance** lets once class get all definitions from another for free
  - + child-class may add or modify definitions
  - – but we can't remove definitions
- Defines a parent-class / child-class relationship.
- Defines a **subtype relationship**.

Why do we want inheritance?

- powerful **code reuse** mechanism

→ retype as little as possible, always!

- Gives us **subtyping**.

→ allows for specialization

→ one definition works on related types

Re-use benefits

- modifications are centralized
- re-use tested code (don't re-implement)
- contributes to elegance, maintainability

- lets compiler know types are related

→ makes code flexible in a controlled way

Sub-Classes

- class is a type → subclass is a subtype
- subtypes are like subsets:

  Integers: {…,-2, -1, 0, 1, 2, …}

  Naturals: {0, 1, 2, …}

- every Natural value is also an Integer value:

→ Natural is a subset of Integer. " Natural $\subseteq$ Integer "

- We know some subtypes from math:

  Natural $\subseteq$ Integer $\subseteq$ Rational $\subseteq$ Real $\subseteq$ Complex

Example Hierarchies

- Freshmen $\subseteq$ Undergrads $\subseteq$ Students $\subseteq$ People
- SUVs $\subseteq$ Trucks $\subseteq$ Vehicles $\subseteq$ Machines

- A type has a set of values, so a subtype contains a subset of the superset's values.

Identifying Hierarchies

- Look for similarity in structure
- Look for more specific versions of things (Mammal, Primate; Bird, Penguin)

- Some classes only exist as links between other useful classes.
  - Mammal, Truck, Student, Container, etc.
  - some def's could be placed in these intermediate places

- Remember: Classes define **what** data/behavior is common between separate classes.
  - you might still be creating many objects of type Parallelogram and of type Rectangle, but their definitions could share side1 and side2 variables.

Clarification:

extending a class ≠ instantiating object

- Extending a class:
→ making one 'blueprint' from another.
→ no objects created just yet

- Instantiating an object:
→ using class 'blueprint' to make an object
→ Class definitions used, but not made here.
→ objects created

Example

```
public class Person {
        public String name;
        public int age;
}
```

```
public class Student extends Person{
        public int studentID;
}
```

- A Person object has 2 instance variables: **name**, **age**
- A Student object has 3 instance variables: **name**, **age**, **studentID**
- the Student class is a **subclass** of the Person class.

Example - constructors

```
// in Person class:
public Person (String name, int age){
      this.name = name;
      this.age = age;
}
```

```
// in Student class:
public Student (int id, String name, int age){
      super(name,age); // call parent constr.
      this.studentID = id;
}
```

Constructor Notes

- Constructors are never inherited.

- child constructor MUST call a parent constructor as first instruction
- feed it expected arguments, use the name super()
- Java inserts **super()** as implicit first instruction if you don't first call super() yourself.
  - requires parent class with no-param constructor.

Constructors

- constructor chaining ensures parent class still has total control over all objects that may be treated as that type
  - child objects usable where parent type needed
  - child objects can't create inconsistent states
  - child objects can't violate parent class's permissions (public/private/<default>/protected)
  - every child object starts as a parent-class object that gets specialized.

The **super** Reference

- **super** is used by child to name members inherited from parent.
  - calling parent constructor
  - calling other methods from parent
  - accessing (shadowed) fields from parent

Pytania Poll

- Inheritance

Inheritance and Visibility

What can a child class see?

If a parent class declares a member as:

- **public**   anyone can see these, so children can too.
- **private**: not even children get to see these members.

  (They are still inherited, though!)

- **protected**: children can also see, though nobody else outside of the package can.
- <default>: if child is in the same package, yes.

Multiple Inheritance (forbidden)

- Java only supports ***single inheritance***, meaning a derived class can have only one parent class

- ***Multiple inheritance*** allows a class to be derived from two or more classes, inheriting the members of all parents (other languages do this)

- name collisions (between both parents) have to be resolved.

→ Java's *interfaces* exhibit multiple inheritance!

Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own

- The child's method must have the same signature as the parent's method, but can have a different body

- object type determines which version is invoked
→ child runs its version, parent runs its version
→ *object type determines it, not variable type!*

Overriding Methods

- tailor the functionality of child class to the particular type.

- Example
  - Base class **Animal** has a method **makeNoise()**

  - Child class **Dog** implements (overrides) **makeNoise()** to print "woof!"
  - grandchild class **ScottishTerrier** then might print "woof at ye, scunner!"
- we must have the same signature for **makeNoise()**

Overriding Methods

- A method in the parent class can be invoked explicitly using the **`super`** reference

- If a method is declared with the **`final`** modifier, it cannot be overridden.

- fields can also be overridden (redefined)

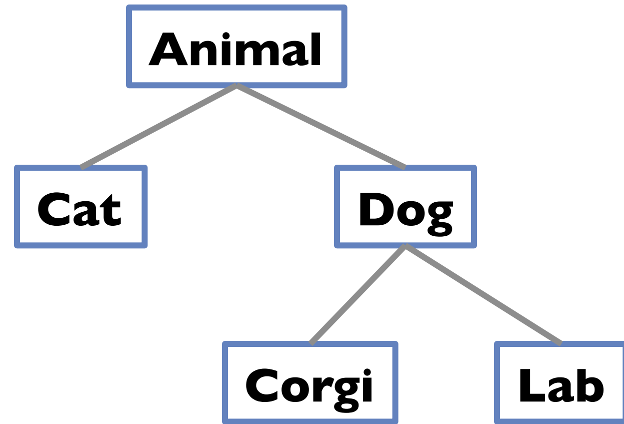
→ called ***shadowing variables***. And it's usually a bug.
  - avoid – we can lose access to ancestors' versions

Overloading vs. Overriding

- Overloading: methods with same name in same class (perhaps inherited), but with different signatures
- Overriding: two methods, one in parent class and one in child class, with same signature. Child replaced what it inherited.

- Overloading: defines similar operation in different ways
- Overriding: defines same operation in different way for child class

Class Hierarchies



- Two children of the same parent are *siblings*
- push common features as you reasonably can (more reuse)

- child inherits from all its ancestor classes transitively

- no single class hierarchy is appropriate for all situations

The Object Class

- A class, **Object**, is defined in **java.lang**

- All classes derive from the Object class
→ If no parent class specified, Object is used.

- Therefore, the Object is the ultimate root of all class hierarchies.

The Object Class

- Some methods inherited from Object:

String toString ()

→ commonly overridden

→ provides that Classname@address default string.

boolean equals(Object other)

→ Object's version uses **==** (memory location)

→ overridden to specialize for our classes.

specialize: boolean equals (**Person** other)

Abstract Classes

- An ***abstract class*** is a placeholder in a class hierarchy that represents a generic concept

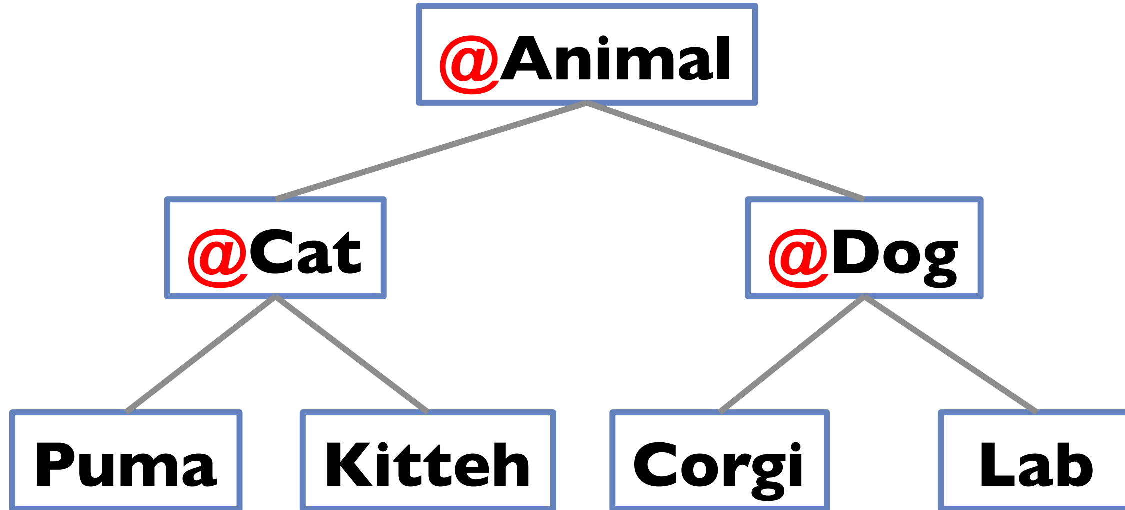
→ An abstract class cannot be instantiated

→ abstract classes can be extended.

- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Product{ … }
```

Abstract Classes



- push common def'ns up to abstract classes
  - even ones w/o implementation (abstract methods)
- gives common type to describe all the child classes

Abstract Classes

- may contain **zero or more** **abstract methods**

→ use **abstract** modifier

→ have no body: replace **{…}** with **;**

→ abstract methods only exist in abstract classes

- may contain non-abstract ("concrete:") methods

- abstract method cannot be defined as **final** or **static** (useless – why?)

Abstract Classes' Children

- **children inherit abstract methods**, too!

  → but, still unimplemented

- child must either override inherited abstract method, or also be declared abstract.

abstract: like a contract

- contract: all objects usable at this type have method impl.

- concrete child class fulfills contract by overriding all abstract things → all methods are available; object usable at the abstract class type

- sometimes a child class is also abstract; doesn't have to fulfill the contract. (Leaves it to later generations)

Abstract class example

abstract class Item represents buyable items

- All items have a barcode (and showBarcode() method)
- But Items are otherwise very different:

→ have abstract method to generate HTML for the item

→ Some have images, some are linked to others

→ method implementations are specific to each item

Visibility, Revisited

- All members of a parent class, ***even private members***, are inherited by children
  - but can't be referenced by name in the child class
- However, inherited private members exist and can be used indirectly!
  - through visible inherited methods

Inheritance Design Issues

- Every derivation should be an ***is-a*** relationship

→ a Student is-a Person; a Penguin is-a Bird.

- Find common characteristics of classes and push them as high in the class hierarchy as appropriate
- Override methods as needed to tailor functionality of a child
- Add new variables to children, but never shadow inherited variables!

Inheritance Design Issues

- each class should manage its own data
- always override general methods such as **`toString`** and **`equals`**
- use abstract classes to connect classes as needed
- use visibility to provide *minimum* access needed

Restricting Inheritance

- `final` class: prohibits extending it
→ a class can't be abstract and final – useless!

- `final` method: can't be overridden by children
→ children are stuck with this version.

- allows parent to guarantee how it's used