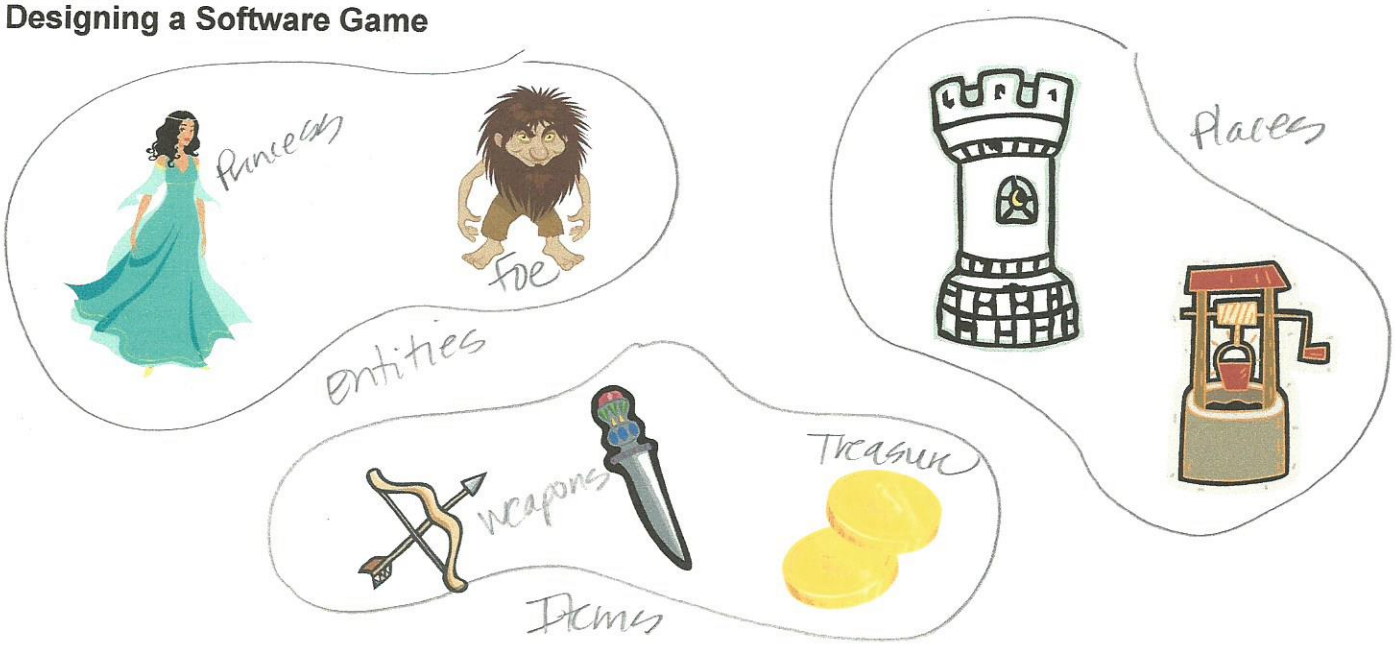
AP Computer Science
Chapter 7 Notes – Inheritance

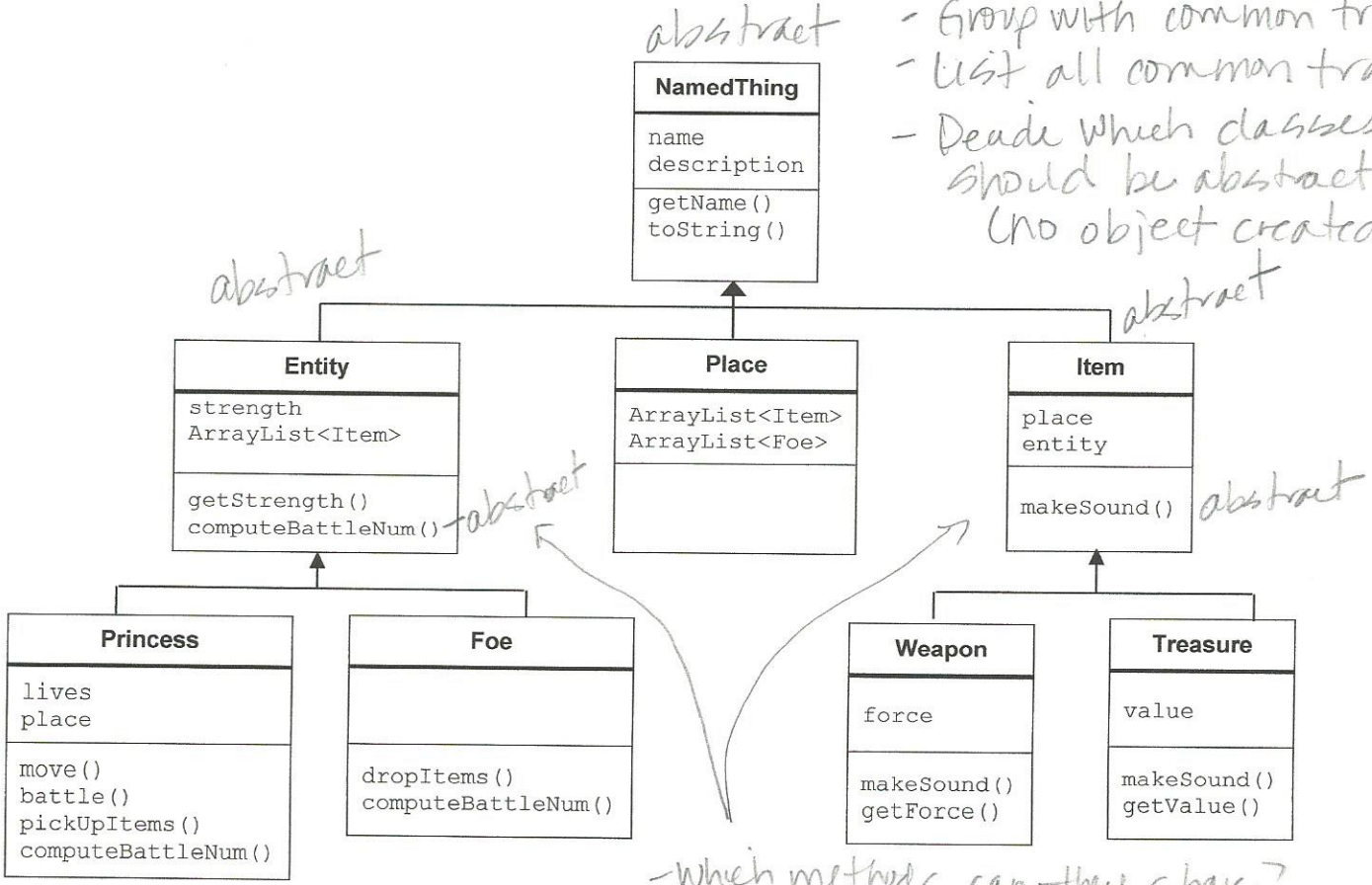
Name Key Per ____

Designing a Software Game



7.1 Create a Cass Hierarchy

- List all nouns in game
- Group with common traits
- List all common traits
- Decide which classes should be abstract - (no object created)



- Which methods can they share?
- which should we override?
- make those abstract if it makes sense (no code - header only).

How will Princess be created? Which parameters need to be passed to constructor?

7.2 Inheritance

Inheritance allows a software developer to derive a new class from an existing one. The existing class is called the *parent class,* or *superclass* or *base class*. The derived class is called the *child class* or *subclass*. As the name implies, the child inherits characteristics of the parent. That is, the child class inherits the methods and data defined for the parent class. To tailor a derived class, the programmer can add new variables or methods, or can modify (*override*) the inherited ones. *Software re-use* is at the heart of inheritance.

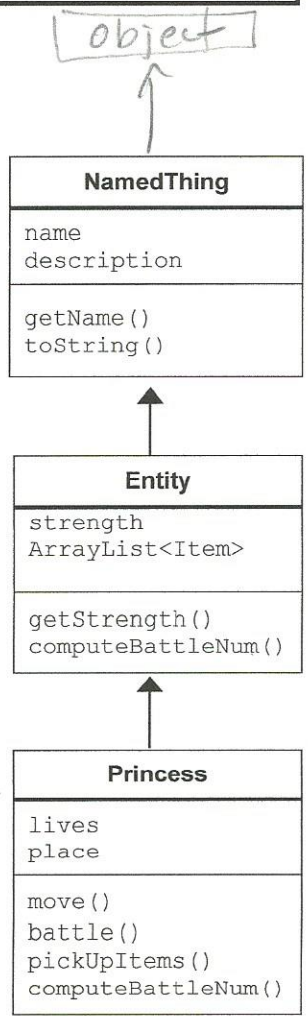
> The main point of inheritance is to allow for code reuse.

All Java classes are derived, directly or indirectly, from the `Object` class.

- The `toString` and `equals` methods are defined in the `Object` class and therefore inherited by every class in every Java program. — Override in your classes.

* every object IS-A Object.

Inheritance relationships often are shown graphically in a UML class diagram, with an arrow pointing to the parent class. Inheritance should create an *IS-A relationship*, meaning the child *is a* more specific version of the parent. In Java, we use the reserved word `extends` to establish an inheritance relationship.



Visibility modifiers determine which class members can be used by derived classes and which cannot. Variables and methods declared with `public` visibility can be used. Variables and methods declared with `private` visibility cannot. The child class has inherited `private` members, but cannot access them directly. Access is through `public` getter and *setter* methods. Inheritance is between *classes* not between *objects*.

Multiple Inheritance

Java is a *single* inheritance programming language. A child class can only have *one* parent. Some languages allow multiple inheritance. What problem might this create?

Which code will be executed for duplicate methods?

This problem is sometimes called the Deadly Diamond of Death (DDD). One way to inherit from different classes is to implement several *interfaces*. Interfaces do not provide any code; therefore if two interfaces have the same method name, the class implementing the interfaces will provide one coding solution. Remember, in Java a class can **implement** many interfaces, but cannot **extend** more than one class.

Overloading vs Overriding

A child class can *override* the definition of an inherited method in favor of its own. The new method must have the same *signature* as the parent's method, but can have a different body. The type of the object executing the method determines which version of the method is invoked.

Don't confuse the concepts of overloading and overriding:

- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

---

7.3 `abstract` class

An abstract class *cannot* be instantiated. It represents a concept on which other classes can build their definitions. A class created from an abstract parent must override all of its parent's `abstract` methods, or the derived class will also be considered `abstract` (and therefore cannot be instantiated).

Can `static` methods be abstract? Remember, `static` methods can be invoked using the class name without declaring an object of the class.

cannot create NamedThing objects
new NamedThing(...);

```
public abstract class NamedThing {
   private String name;
   private String description;

   public NamedThing (String n, String d) {
      name = n;
      description = d;
   }

   public String getName() {
      return name;
   }

   public String toString() {
      return name + " " + description;
   }
}
```

Subclasses cannot access these directly - only through getter/setter methods

will be invoked through subclass constructor.

Subclasses will inherit these - can invoke as if written directly in subclass.

7.4 super Reference

A *constructor* for a child class always starts with an invocation of one of the constructors in the parent class. If the parent class has several constructors then the one which is invoked is determined by matching *argument lists,* parameter type and order.

- A child's constructor is responsible for calling the parent's constructor.
- The first line of a child's constructor should use the `super` reference to call the parent's constructor.
- The call to `super()` comes first, even if you don't write it in. If the parent does not have a no-argument constructor, then using this "shorthand" causes a syntax error.

no Entity objects.

```
import java.util.ArrayList;

public abstract class Entity extends NamedThing {

   private int strength;
   private ArrayList<Item> items;

   public Entity (String nm, String desc, int s) {
      super (nm, desc);
      strength = s;
      items = new ArrayList <Item>();
   }

   public int getStrength() { return strength; }

   public abstract int computeBattleNum();
}
```

* calls NamedThing constructor.

Super calls the parent Constructor MUST be first line in constructor

must provide implementation in subclass.

The `super` reference can also be used to reference other variables and methods defined in the parent's class. Override the inherited `toString()` method inherited from `NamedThing` to include the `name`, `description` and the list of items an `Entity` has.

```
public String toString() {
   String temp = super.toString(); // adds name & description
   for (Item i : items)
      temp += i.toString() + "\n"; // loop through
                                   // items array.
   return temp;
}
```

Inheritance Example

```
public class P {
   private int x;

   public P() {
      x = 10;
   }

   public String one(){
      return x + " P class one() " + two();
   }

   public String two(){
      return "P class two() " + x;
   }

   public String toString(){
      return "P class toString() " + x + " " + one();
   }
}
```

10
x

```
public class Q extends P{

   private int x;

   public Q() {
      x = 7;
   }

   public String two(){
      return "Q class two() " + x;
   }

   public String toString(){
      return "Q class toString() " + x + " " + super.toString();
   }
}
```

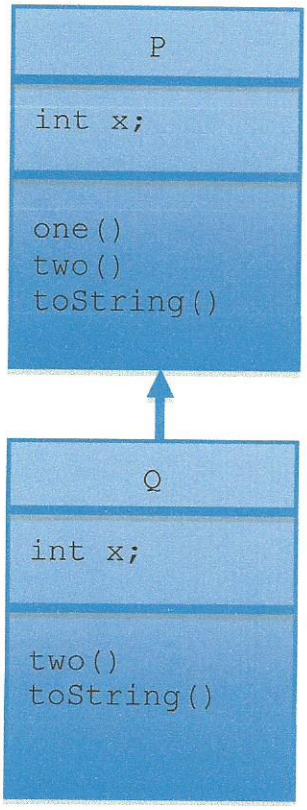
7
x

* variables go with class you are in
* methods always look in object type class first!

```
public class Test {

   public static void main (String[] args){

      P p = new P();          // object created from P -
      System.out.println (p);

      Q q = new Q();          // object created from Q - always look
      System.out.println (q); // here first for method being called

   }
}
```

Output

P class toString() 10 10 P class one() P class two() 10
Q class toString() 7 P class toString() 10 10 P class one() Q class two() 7



Rules:

- Java will look for the method name in the class of the object type first. If it doesn't exist, then Java will look in the parent class. This is true even when the executing method is in the parent class. Java will always look to see if the method exists in the objects class first.

- The variables referenced in a method are the instances in the same class as the actual method code being executed. For example, if a `Q` object is executing the code for method `one()` in class `P`, then the `x` is referring to the variable `x` in the `P` class. Variables go with the class where the method was written.

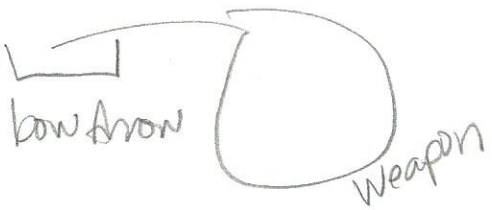
7.5 Polymorphism

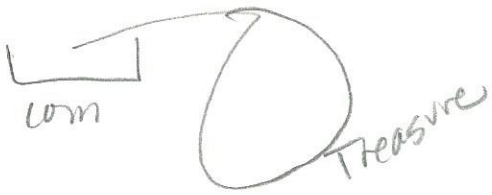
Up to this point, the type of a reference variable has always matched the class of the object to which it refers:

reference var. / object created from this class.

```
Weapon bowArrow = new Weapon ("Cross Bow", "Powerful Cross Bow", princess, 13);

Treasure coin = new Treasure ("Gold Coin", "Shiny Gold Coin", courtyard, 20);
```

ref var type / ref var name. (holds address to object) / object type.






```
public abstract class Item extends NamedThing {

   private Place place;
   private Entity entity;

   public Item (String nm, String desc, Place p) {
      super (nm, desc);
      place = p;
   }

   public Item (String nm, String desc, Entity e) {
      super (nm, desc);
      entity = e;
   }

   public abstract void makeSound ();
}
```

```
public class Weapon extends Item {

   private int force;

   public Weapon (String nm, String desc, Entity e, int f){
      super (nm, desc, e);
      force = f;
   }

   public void makeSound() {
      System.out.println ("Fight, fight, fight!");
   }

   public int getForce() {
      return force;
   }
}
```

calls Item constructor

overrides method from Item

new method just for Weapon objects.

```
public class Treasure extends Item {

   private int value;

   public Treasure (String nm, String desc, Place p, int v){
      super (nm, desc, p);
      value = v;
   }

   public void makeSound() {
      System.out.println ("Cha-ching $$$$");
   }

   public int getValue() {
      return value;
   }
}
```

A reference can be *polymorphic* which can be defined as "having many forms". It is the type of the *object* being referenced, not the *reference* type, that determines which method is invoked.

```
Item someItem = new Weapon ("Cross Bow", "Powerful Cross Bow", princess, 13);
someItem.makeSound();
someItem = new Treasure ("Gold Coin", "Shiny Gold Coin", courtyard, 20);
someItem.makeSound();
someItem.getValue();
```

Weapon IS A Item, not Item is-A weapon

← invokes object type method (Weapon

← invokes Treasure implementation

← ERROR!

Fight, Fight, fight

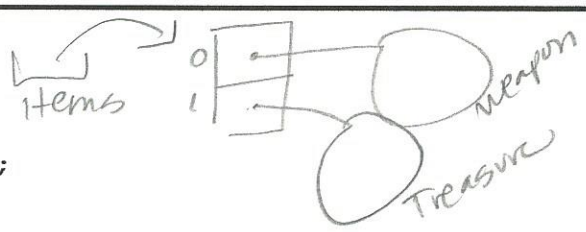
v. method must be in Ref var type to ensure it is in object type.

```
((Treasure) someItem).getValue(); // cast
```

A reference variable can refer to any object created from any class related to it by inheritance (by extending a class or implementing an interface). A class name or interface name can be used as the type of a reference variable.

* Ref var type can be class or interface

A polymorphic reference uses the type of the object, not the type of the reference, to determine which version of a method to invoke.




```
ArrayList<Item> items = new ArrayList<Item>();

items.add(new Weapon ("Cross Bow", "Powerful Cross Bow", princess, 13));
items.add(new Treasure ("Gold Coin", "Shiny Gold Coin", courtyard, 20));

items.get(0).makeSound(); // fight, fight, fight
items.get(1).makeSound(); // cha ching!!
```

Item has makeSound()

* Weapon and Treasure IS-A Item.

Which of the following polymorphic statements are legal? Assume there is code not shown and the following statements are executed in the order presented and appear after the preceding statements:

no `((Treasure)items.get(0)).getValue();` // error can't cast Weapon to look like Treasure - Weapon ~~IS-A~~ Treasure doesn't work!

no `items.get(0).getForce();` // getForce not in Item

yes `Entity dragon = new Foe("Harold", "Dragon", 10);` // Foe is-A Entity

IS-A? No.

no `Item courtyard = new Place("courtyard", "sunny place");` // Place IS-A Item doesn't work.

yes `ArrayList<NamedThing> gamePieces = new ArrayList<NamedThing>();`

yes `gamePieces.add(new Foe("Shrek", "Ogre", 5));` // Foe is-A namedThing - yes.

yes `gamePieces.add(new Treasure ("Ruby", "Jewels", dragon, 20));` // Treasure is-A NamedThing - yes

no `gamePieces.get(1).makeSound();` // makeSound() not in NamedThing

// can cast as Item or Treasure to correct.

Binding

At some point, the computer has to execute the code to carry out a method invocation. This is called *binding* a method invocation to a method definition. Most of the time binding happens at *compile* time. For polymorphic references, binding happens at *runtime*.