- Previous Lecture:
  - One constructor calling another
  - Overriding methods
- Today's Lecture:
  - Using super to access members from the superclass
  - Polymorphism
  - **`Object`** class
  - **`Abstract`**
- Reading:
  - Sec 11.8

Accessing members in superclass

super

- From constructor in subclass, call superclass' constructor
- Access superclass' version of a overridden method. E.g.:

super.toString()

`static` methods & variables

- Do not re-declare `static` components!
- Same rules for inheritance (accessibility) with respect to visibility modifiers
- Static method: implicitly **`final`**
- Static variable: same memory space as superclass

Important ideas in inheritance

- Single inheritance
- Keep common features as high in the hierarchy as reasonably possible
- Use the superclass' features as much as possible
- "Inherited" ⇒ "can be accessed as though declared locally"
  (**`private`** variables in superclass *exists* in subclasses; they just cannot be accessed directly)
- Inherited features are continually passed down the line
- Use different hierarchies for different problems

Polymorphism

- "Have many forms"
- A *polymorphic* reference refers to different objects (related through inheritance) at different times

Suppose class Plane extends Vehicle

```
Vehicle mover; //a Vehicle reference
Plane flyer;   //a Plane reference
mover= new Vehicle(...);
flyer= new Plane(...);
// A plane is a vehicle
   mover= new Plane(...);
   mover= flyer;
// A vehicle is not a plane
   flyer= new Vehicle(...); //invalid
```

Another polymorphic example

```
Vehicle[] mover = new Vehicle[5];

mover[0]= new Vehicle(...);
mover[1]= new Plane(...);
mover[2]= new Plane(...);
mover[3]= mover[1];
```

The reference type may not be the same as the object type!

Accessing methods/variables through a polymorphic reference

Dice d= new TrickDice(...);

Consider the reference type and object type:

1. Which type determines whether a method/variable can be accessed?
2. For an overridden method, which type determines which version gets invoked?

Accessing methods/variables through polymorphic references

The ***type of the reference*** determines the methods and fields that can be accessed

```
class V {
  public int num1;
  public void vmethod() { num1++; }
}
class W extends V {
  public int num2;
  public void wmethod() { num2++; }
}
```


Client code:

```
V x= new W();
System.out.println(x.num1);  //valid?
System.out.println(x.num2);  //valid?
x.vmethod();  //valid?
x.wmethod();  //valid?

System.out.println( ((W) x).num2 );
((W) x).wmethod();
```


Client code:

```
V x;  // x references type V or its subtype
System.out.print("Which type, V or W? ");
Scanner keyboard= new Scanner(System.in);
char input= keyboard.nextChar();
if (input=='V')
   x= new V();
else
   x= new W();

System.out.println(x.num1);  //?
System.out.println(x.num2);  //?
x.vmethod();  //?
x.wmethod();  //?
```


Accessing methods/variables through a polymorphic reference

Dice d= new TrickDice(...);

Consider the reference type and object type:

1. Which type determines whether a method/variable can be accessed?

   reference type
2. For an overridden method, which type determines which version gets invoked?

   object type

Accessing *overridden* methods through polymorphic references

- The ***type of the object*** determines which version of the method gets invoked
- Class `Dice` has method `roll` that class `TrickDice` overrides:

```
Dice d1= new Dice(...);
Dice d2= new TrickDice(...);
d1.roll(); //Dice's version
d2.roll(); //TrickDice's version
```


instanceof

- instanceof is an operator for determining when an instance is of (from) a particular class
- See example in class House

The `Object` class

If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class

⇒ All classes are derived from the **`Object`** class

`class Room`

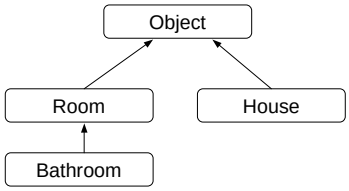
is the same as

`class Room extends Object`

The `Object` class

- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class

  ⇒ All classes are derived from the **`Object`** class



The `Object` class

- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class

  ⇒ All classes are derived from the **`Object`** class

- `toString`: "default" instance method defined in the `Object` class
- Arrays are `Objects`, literally!

`abstract` class

- A placeholder in a class hierarchy that represents a generic concept
- Cannot be instantiated
- Modifier: `abstract`

```
public abstract class Geometry
```

- Can contain abstract methods

```
public abstract double Area();
```

- Subclasses of abstract classes will "fill out" these abstract methods

```
/* A Room has an id number and a messiness level */
class Room {
  private static int nextID = 1; //id of next room to be
                                 //created
  protected int id; //room number
  private int mess; //mess level

  /** A Room has unique id and messiness level mess */
  public Room(int mess) { this.mess = mess;   id = nextID;   nextID++; }

  /** = String description of this Room */
  public String toString() { return "Room " + id; }

  /** Reduce mess by 1 but keep mess>=0 */
  public void clean() { mess--;
                        if (mess<0)  mess=0; }

  /** Print status of Room */
  public void report() { System.out.println(toString() +
                         ", has mess level " + mess);     }

  /** Print how many rooms have been created */
  public static void countRooms() {
    System.out.println((nextID-1)+ " rooms in total"); }
} //class Room
```

```
/* A Bathroom is a Room and may have a shower */
class Bathroom extends Room {

  private boolean hasShower; //=has a shower

  /** A Bathroom has initial mess level, boolean hasShower */
  public Bathroom(int mess, boolean hasShower) {
    super(mess);
    this.hasShower= hasShower;
  }

  /** = String description of this Bathroom */
  public String toString() {
    String line= super.toString();
    line += ", a bathroom";
    if (hasShower)   line += " with a shower";
    return line;
  }

  /** Clean repeatedly.  Call method clean four times */
  public void majorCleanUp() {
      clean();  clean();  clean();  clean();
  }
} //class Bathroom
```

```
public class House { //see online version for more examples
  public static void main(String[] args) {

    Room[] rooms= new Room[5];

    for (int i=0; i<rooms.length; i++)
      if (Math.random()< 2.0/3) //{twice as likely to be Room than Bathroom}
        rooms[i]= new Room(10);
      else
        rooms[i]= new Bathroom(20,true);
  }
} //class House
```